University of Alberta

Heuristic Search Applied to Abstract Combat Scenarios

by

Alexander Kovarsky

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-95786-1
Our file Notre référence
ISBN: 0-612-95786-1

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

To my parents,

Emma and Oleg

Acknowledgement

I would like to express my gratitude to my supervisor, Dr. Michael Buro, for his help, great advice as well as for financial support during the course of this research. His genuine interest and expertise in this research as well as in the areas of heuristic search and real-time strategy gaming in general were among my main reasons for deciding to pursue this research and for subsequently achieving its goals.

Table of Contents

1. Introduction 1

1.1 The Field of Artificial Intelligence 1

1.2 Games as a Test-bed in Artificial Intelligence 3

1.3 Real-Time Strategy Games 4

1.4 Open Real-Time-Strategy Test-Bed (ORTS) 6

1.5 Proposed Approach for RTS Research 7

1.6 Abstract Combat Scenarios 9

1.7 Contributions 11

2. Related Work 13

2.1 Lanchester Equations Example 15

3. Problem Description 17

3.1 Basic Problem Set-Up 19

3.2 Defensive Actions Modification 20

3.3 Set-Up with Ranged Attacks 22

3.4 Abstract Combat Examples 23

3.4.1 Unlimited Range Example 23

3.4.2 Range Example 27

4. Abstract Combat Decision Algorithms 28

4.1 Motivation 29

4.1.1 Examination of Problem Complexity 31

4.1.2 Dependence in Games with Simultaneous Moves 32

4.1.3 Tackling The Challenges of the Domain 33

4.2 Implemented Algorithms 34

4.2.1 Linear Programming 34

4.2.2 Alpha-Beta (AB) 37

4.2.3 Random Alpha-Beta (RAB) 38

4.2.4 Monte Carlo 43

4.2.5 Random Player 45

4.3 Evaluation Functions 46

4.3.1 Simple Evaluation Function 47
4.3.2 Square Root Evaluation Function 48
4.3.3 Tournament Evaluation Function 49
4.4 Other Improvements and Applications 50
4.4.1 Selective Search 50
4.4.2 Removing Duplicates 51
4.4.3 Delayed Move Execution 51
4.4.4 Balancing Scenarios Application 52
5. Experiments 54
5.1 Experiments Preview 54
5.2 Experimental Setup 56
5.3 Results of Experiments 59
5.3.1 Experiment 1 (Scoring Setting for RAB and Monte Carlo) 59
5.3.2 Experiment 2 (Performance of All Methods) 60
5.3.3 Experiment 3 (Evaluation Function Selection) 63
5.3.4 Experiment 4 (Move Selection Experiment) 65
5.3.5 Experiment 5 (Varying RAB's Nodes vs Constant AB) 68
5.3.6 Experiment 6 (Strict Constraints) 70
5.3.7 Experiment 7 (Degree of Move Interdependence) 72
5.3.8 Experiment 8 (Sticking to Target Improvement) 74
5.3.9 Experiment 9 (Examining the Range Expansion) 75
5.4 Unit Value Balancing 79
6. Conclusion 80
7. Future Work 81
8. References 86

List of Tables

Table 1: Shows the number of Y survivors and casualties as the number of initial y's units increases 17

Table 2: Predefined ranges of hitpoints, attack values and cooldown periods for units used in our experiments..... 57

Table 3: Shows the results for team 2 after balancing for non-defensive (a) and defensive scenarios (b) 80

List Of Figures

Figure 1: Shows the LPs for two players for computing Nash-optimal strategies. Player "Max" has moves from $1 \ldots n_1$, while "Min" is from $1 \ldots n_2$. $A_{i,j}$ is the score from "Max"'s perspective if a pair of moves (i, j) is chosen. x_i and y_j are probabilities of choosing moves i and j, for "Max" and "Min", respectively. 35

Figure 2: Pseudo-code for the LP player's algorithm 36

Figure 3: Shows a sample RAB tree. The black player moves first at the root followed by the white player. At the next level, the player to move is randomly selected. This player's move is followed by the move of the opponent of the randomly selected player. 40

Figure 4: Randomized binary minimax tree (a) and alternating binary minimax tree (b). Both trees have the same structure, number of nodes and exactly the same values at leaf nodes. The difference is in the order at which the moves are executed and the value that is propagated to the root. 40

Figure 5: Pseudo-code for the RAB algorithm 41

Figure 6: High-level pseudo–code of the Monte Carlo method 44

Figure 7: Win ratio of RAB over AB and Monte Carlo over AB for different scoring settings. 60

Figure 8: Win ratio for each individual method in a round robin tournament for a non-defensive scenario. 61

Figure 9: Results of all methods playing against one another in a non-defensive scenario. 62

Figure 10: Win ratio for each individual method in a round robin tournament for a non-defensive scenario. 62

Figure 11: Results of all methods playing against one another in a defensive scenario. .. 63

Figure 12: Evaluation function performance: (a) non-defensive scenarios (b) defensive scenarios. 64

Figure 13: Move selection results in 3 versus 3 non-defensive (a) and defensive (b) scenarios. Results show that concentrating effort on a subset of moves payoffs..... 66

Figure 14: Move selection results in 4 vs 4 non-defensive (a) and defensive (b) scenarios. Results show that concentrating effort on a subset of moves payoffs..... 67

Figure 15: Results of RAB wins over AB as the number of RAB nodes is increased from 20,000 to 400,000 and AB's node count is fixed. The results are shown for non-defensive scenarios. 69

Figure 16: Results of RAB wins over AB as the number of RAB nodes is increased from 50,000 to 400,000 and AB's node count is fixed. The results are shown for defensive scenarios. 69

Figure 17: Results of RAB wins over AB as the number of nodes is increased from 10,000 to 100,000 for 3vs3 scenarios and from 20,000 to 200,000 for 4vs4 for non-defensive scenarios. 70

Figure 18: Results of RAB wins over AB as the number of nodes is increased from 30,000 to 200,000 for 3vs3 scenarios and from 50,000 to 400,000 for 4vs4 for defensive scenarios. 71

Figure 19: The ratio of wins of RAB over the AB for 3 vs 3 scenarios as the move dependency increases from no defensive action to high probability of a defensive action. The figure also shows the number of wins/losses as a percentage of games played in such situation, indicating result variance 73

Figure 20: The ratio of wins of RAB over the AB for 4 vs 4 scenarios as the move dependency increases from no defensive action to high probability of a defensive action. The figure also shows the number of wins/losses as a percentage of games played in such situation, indicating result variance 73

Figure 21: The percentage of wins for RAB in 3 vs 3 and 4 vs non-defensive scenarios as the number of nodes given is increased from 10,000 to 800,000. 75

Figure 22: The combined totals of win ratio achieved over all methods for each individual method in non-defensive scenarios with range. 77

Figure 23: Results for ranged non-defensive scenarios in 3 vs 3 situations 77

Figure 24: The combined totals of win ratio achieved over all methods for each individual method in non-defensive scenarios with range. 78

Figure 25: Results for ranged non-defensive scenarios in 4 vs 4 situations 78

1. Introduction

1.1 The Field of Artificial Intelligence

The field of artificial intelligence is a vast area, which is comprised of dozens of sub-fields. Therefore, providing a single or even adequate definition of what is artificial intelligence is a difficult, if not an impossible task. Nevertheless, there are several common goals that are relevant to most sub-fields of artificial intelligence. Trying to understand intelligent entities is a goal artificial intelligence shares with sciences that study humans, such as philosophy and psychology. However the main goal of artificial intelligence is to build intelligent systems and subsequently to study them, because of the key belief that much can be learned from such systems and the results they produce.

There are many definitions of AI present in the current literature. In their Artificial Intelligence textbook [28], Russell and Norvig divide the definitions of AI according to two criteria. One criterion separates the definitions that are based on thought-processes versus the ones that address behaviour. The other separates between the measures of success in terms of human performance as opposed to rationality. Therefore, all definitions of artificial intelligence according to [28] fall into four main categories:

1. Systems that think like humans
   - "The exciting new effort to make computers think … *machines with minds*, in the full and literal sense" [16].
2. Systems that think rationally
   - "The study of computations that make it possible to think reason and act" [40].
3. Systems that act like humans
   - "The study of how to make computers do things at which at the moment people are better" [27].
4. Systems that act rationally

- "The branch of computer science that is concerned with the automation of intelligent behavior" [23].

The broad range of definitions trying to define what AI is or what it should become suggests not only that artificial intelligence is still in its infancy and most scientists are still not completely sure where it is going, but also that there are many different sometimes seemingly unrelated areas that artificial intelligence tries to encompass.

The main areas of artificial intelligence include: *agent theory*, which studies the properties of intelligent agents and the interactions between them; *machine learning*, which studies ways to adapt to new situations and learn from previous behaviours; *heuristic search*, which uses computers' high computation potential to find solutions to complex problems; *knowledge representation*, which investigates ways to represent and store knowledge in intelligent systems; *logical reasoning*, which looks into ways computers should approach and solve different problems; *natural language processing*, which researches ways for computers to understand and communicate in a particular human language.

Each of the above sub-fields standalone are still very broad areas and do not provide a precise description of a specific research. For example, the area of machine learning is further subdivided into areas such as reinforcement learning, neural networks, genetic algorithms, etc. Furthermore, the borders that separate different sub areas are not very well defined or "fuzzy". Often it is difficult to determine whether a particular project belongs to one or another of the many sub-fields of artificial intelligence and on occasion some research can be justifiably claimed to belong to two or more such sub-fields at the same time.

The research described here is a good example of this. The techniques used for this research belong to the area of heuristic search. The test domain for the research is that of game-playing, specifically commercial computer games. But, the research also deals with real-time AI issues as well as game theoretical solutions to simultaneous move games.

1.2 Games as a Test-bed in Artificial Intelligence

The research in games started more than 40 years ago by the pioneers of artificial intelligence Arthur Samuel, Claude Shannon and Alan Turing. In [31][36][29] they introduced the concepts that are still behind many of the modern high-performance game-playing programs. Those concepts have been successfully applied to create strong game playing programs in chess [18], checkers [30], Othello [7], backgammon [2], and poker [4]. The main goal of creating strong game playing programs, however, is not to improve our knowledge of those games, but to explore new and exciting research ideas. Indeed, many advances in the area of search were discovered as a result of building game playing programs. Most successful game-playing programs require the use of those methods developed by those pioneers; ours is not an exception

Nevertheless, advances made in playing the games themselves cannot be completely ignored. Successful programs that can beat the best humans increase the awareness about advances in artificial intelligence in the general public and thus generate more interest in the subject. There are many examples such successes in the games of chess, checkers, backgammon and Othello. The most prominent of these successes is the 1997 victory by IBM's Deep Blue chess playing computer over then world champion Gary Kasparov, which generated a lot of interest in the general public, but more importantly helped bring AI research into mainstream. There were other major successes including the world champion checkers program Chinook [30], developed by University of Alberta's Jonathan Schaeffer.

So the questions of why such AI success was achieved in board games and why games are excellent domains to pursue artificial intelligence research naturally arise. The main reason is that games provide a well-defined environment in which algorithms can be tested, polished, and evaluated. In turn, those algorithms can be applied to help solve many real-life problems. Therefore, tackling a simpler and narrower problem in a very restricted domain and then trying to apply what was learned from it into a more difficult

realistic environment is preferable to trying to tackle more difficult problems at once without having any intuition about them.

1.3 Real-Time Strategy Games

Commercial computer games are quickly becoming a major test-bed for artificial intelligence research. In the past most computer game companies concentrated their efforts on improving the game graphics, without putting much emphasis on the game AI. This was justifiably so since at that time better graphics were the main selling point for most games. This trend has changed recently as most games have reached very high-levels in graphics and gamers (specifically real-time strategy gamers) are starting to stress the often-unintelligent AI behaviour in most games. Also as processors become faster every year, more cycles can be used for artificial intelligence computations in computer games. This is an exciting development for AI research since the domain presents many challenges that are not found in traditional AI research in games.

In games such as chess, where most research in games was done, there is a complete knowledge of the state, little restriction placed on the time given to make a decision, and only two search agents. However, in most real-life problems there is only a partial knowledge of the environment, the environment and the goal state are dynamic in nature, decisions have to be made in real time, and there are multiple agents.

Commercial computer games, however, have more elements that exist in real-life problems, such as strict time constraints. In addition, there is a demand for AI that can provide good solutions under such restrictions. Therefore, such games are ideal applications for real-time AI research.

While board games such as chess and checkers are mostly interesting as test-beds for AI research, commercial computer games are a huge industry by itself. According to a Time

magazine article [34], the computer games industry is twice as big as the movie industry in the United States, having annual revenue of 21 billion US dollars. Thus, research performed in such games can be directly and quickly applied to improve games in an industry aching for better AI solutions, due to the increasing criticism of often primitive game AI behavior.

Real-time strategy (RTS) games will be used as a test-bed for our research, since such games not only have many of the challenges present in the real world, but also provide a well-defined environment in which methods can be investigated and perfected. In RTS games players compete for resources, scattered over a terrain, by setting up an economy, building armies, and guiding them into battle in real-time. RTS games offer a greater variety of fundamental artificial intelligence research problems, as compared to other genres of commercial computer games. Some of the challenges found in most real-time strategy games are [11]:

1. Resource management – players gather resources to build infrastructure, attack forces and defence structures. Players need to balance the way they invest in different areas.

2. Decision making under uncertainty – players often have to make decisions based on incomplete information. For example, they might not have complete knowledge of the map or exact knowledge of enemy forces' locations.

3. Spatial and temporal reasoning – static and dynamic terrain analysis, as well as understanding temporal relations of actions, is of utmost importance in RTS games. Current AI in commercial games mostly ignore these issues.

4. Collaboration – In RTS games groups of players can join forces and share intelligence. Coordinating those actions effectively by communication among the parties is a challenging research problem.

5. Opponent Modeling, Learning – the biggest weakness of AI in most RTS games is the inability to learn from mistakes by adapting future behavior. For human players it take a short time to find such weaknesses in game AI and exploit them. This makes games less enjoyable and reduces their re-playability value.

6. Adversarial real-time planning – the state space in a typical RTS game is far larger than in a board game such as chess, thus searching the original state space is impossible in real-time. Therefore, effective abstraction techniques have to be found to make the search space smaller and to hierarchically divide it into more specialized sub-tasks.

1.4 Open Real-Time-Strategy Test-Bed (ORTS)

Although there are many commercial RTS games that exist today, using them as a test-bed is not an option. Most of those games were designed as complex rule-based systems that are difficult to maintain and expand on. Moreover, game companies, in general, are quite reluctant to provide their source code, because of a fear of spying by their competitors. Thus, to be able to work on improving the state of the art of AI in real-time strategy games, a new test-bed that has most of the features that are present in a typical RTS game has to be created from scratch.

Work on the Open-Real-Time-Strategy (ORTS)[8] project is currently being done by the ORTS group at University of Alberta. Its main goal is the creation of hack-free server-side RTS test-bed that incorporates most elements that are present in typical commercial RTS games. Currently, ORTS has an *nxm* grid in which units of both teams are

positioned. To enhance the realism of the simulation, the grid consists of features such as rivers, hills, mountains, as well as realistic views of obstructions (eg. a unit won't be able to see through mountains). In addition, units in ORTS have different attributes such as speed, attacking strength, defensive strength, range of weapons, etc. ORTS allows AI researchers to connect their own client AI software to the ORTS server to compete against other client software.

1.5 Proposed Approach for RTS Research

ORTS provides researchers with an environment, which they can use to test their ideas. How they choose to proceed in creating the AI for their team is open to them. Designing a complex rule based system to tackle the whole game at once, similarly to what is done in most commercial RTS games, is not the best way to approach the problem since such systems have proven to be difficult to maintain and almost impossible to expand. Thus, a hierarchical approach for solving problems in RTS games that uses the best methods available in the areas of machine learning and search can have a great effect on improving the state of the art of AI in such games.

Consequently, research in RTS games can be divided into two branches: higher-level AI and lower-level AI. Higher-level AI refers to the management of resources, decisions on what to build, and strategic decisions on sending units into battle. Lower-level AI refers to the behaviors of a single unit and small groups of units that are given a command. Lower-level AI should be the starting point of any research, since without effective solutions in this area, research on higher-level reasoning and planning cannot proceed.

There are several challenging issues in lower-level AI. One is pathfinding, which refers to finding a route in a grid from a start location to the goal. Another issue is the creation of a stronger AI for a small group of units that have a common goal. An additional problem is finding effective solutions for abstract combat scenarios between two opposing teams

consisting of several units each. Some of the challenges present in each of those areas are as follows:

1.) Pathfinding problem for a single agent

Time is a vital issue for pathfinding in the RTS domain. Consequently, traditional search algorithms, such as A* will not work, because they cannot guarantee a solution within a given time. The A* or IDA* algorithm cannot be interrupted at any time during its search because it needs to complete the search in order to find a solution. Anytime algorithms can be interrupted at any time always producing a solution (Note: A solution produced is not necessarily an optimal solution), while real-time algorithms produce solutions given the constraints of the environment they operate in. Fortunately, there is a family of newer real-time search algorithms available, which can produce near-optimal solutions given a limited time. The two most well-known of these are LRTA*[20] and D*[33]. Scalability is another issue that needs to be addressed, to tackle an expected decline in performance as size of the grid increases. To minimize this decline the search space size for a selected algorithm has to be decreased. This can be done through search space abstraction, by dividing the search space into areas of the grid, instead of searching on individual locations.

2.) Multi-agent coordination for a small group of units

In a typical RTS environment one task is often given to a group of units that belong to several different classes (eg. tanks, soldiers) rather than to a single unit. Each class of units in such a formation has different strengths and weaknesses. The general issue here is how to make units in such a group act in a coordinated way. What types and proportions of units should be in a formation? How the units should be positioned on a grid? When should they decide to abandon a given goal once it is unattainable or too costly to achieve?

3.) Abstract Combat Scenarios

In such scenarios two small groups of units (2 – 4 units each), battle it out in real-time until one group is eliminated. The key challenge here is how to decide on the amount of firepower to direct at each opponent in order to get the most advantage and more importantly how to do this in real-time. Designing effective methods that provide solutions to abstract combat scenarios can be looked at as the first step towards developing advanced pathfinding techniques.

1.6 Abstract Combat Scenarios

The main purpose of this research is investigating and finding effective methods for abstract combat situations, or also known in military literature [15] as *combat attrition* scenarios. Such scenarios have long been a focus of military research. They present many challenges even without considering the real-time constraints. The main challenge is that there is no simple (i.e. cheap to compute) heuristic solution that works in most situations. Such heuristics are used in RTS games and commonly result in unintelligent unit behavior. Therefore, designing a method that can generate good moves for abstract combat scenarios in real-time can improve the combat AI in RTS games.

In a typical scenario two teams consisting of several types of units battle with each other in real-time. All defending units are located within reach of every attacking unit and vice versa. There are several types of units that are used for this research. They can be looked at as units in an army such as tanks, infantry and artillery. Each unit has three attributes: attack value, hitpoints, and cooldown period. Attack value refers to the amount of damage a unit can inflict on its opponents. For example, tanks and artillery will have higher attack values than infantry. Hitpoints refers to the defensive strength of a given unit. For instance, a tank will have more hitpoints than artillery or an infantryman, and

thus will be more difficult to eliminate. Cooldown refers to how often a given unit can fire. An infantryman can fire almost continuously, while it takes time for a tank to cooldown its weapon.

Given a scenario where two teams consisting of several units battle each other, the central issue is deciding which unit should fire at which enemy unit at a given time. The properties of our domain differ from traditional domains where search is used and consequently make it more challenging. One complicating issue is that both teams can fire simultaneously, which makes it seemingly difficult to use an of-the-shelf algorithm such as alpha-beta, since the algorithm is designed for situations where players alternate turns. Second, all units move at the same time at each turn. Therefore, the successors are all possible combinations of all units' moves. This increases the branching factor by a high degree, as compared to traditional board games where one piece usually moves. Another issue is that as a number of units in each team increases, there is an exponential increase in the state space, which results in a significant decrease in performance (Refer to Section 4.1.1). Even though alpha-beta or other real-time approaches can produce a solution given tight time constraints, the quality of such solutions will generally decrease significantly as the number of units increases.

In order to develop suitable algorithms for generating moves in such scenarios, we have created a simulation environment where such games are played out till the end (i.e. when one side is eliminated). In such games we test several approaches for selecting a team's actions in real-time. Two of the approaches are search-based, which use a modified version of an alpha-beta algorithm. In one algorithm, which we refer to as randomized alpha-beta (RAB), teams do not execute moves one after another, but at certain interior nodes there is a probability for either team to go. Other methods are a Monte-Carlo simulation type player and a linear program (LP) player.

Although at first glance creating effective methods for such combat scenarios might not seem very significant, there are a number of important lessons to be learned from it and there are several real-life applications where methods can be applied immediately. First,

this research can determine whether modified two-player heuristic search algorithms can be used effectively in situations with simultaneous move executions, huge state spaces, and severe real-time constraints.

Second, effective real-time algorithms can be immediately incorporated into an RTS game and will be a vast improvement over the currently used rule-based local combat AI. In addition, our methods can be used as an AI helper (for a human player) for resolving small battles that can alleviate the human commanders from playing them out by hand and instead allow them to concentrate on more strategic global decisions. It can also be used as a planning assistant for either a human or a computer player. Having such an assistant would allow a player to simulate an expected outcome of a potential encounter without actually sending units into battle. Such simulations will help the player decide whether to send units to fight or not.

Another problem that we can address is that of creating equal strength teams which consist of different types and numbers of units. This problem is encountered by game designers when designing interesting scenarios for a game. In such scenarios each player should have a different looking starting position that consists of different types and numbers of units, but each position should have an equal probability of winning. A good helper tool would allow scenario designers to play a series of simulated games, which perturb the units' numbers and strengths by a small degree during each simulation turn until the game is a draw. Such tool will be able to automatically generate equal starting positions that are close to positions given as its input. Currently, such tuning is done by hand.

1.7 Contributions

This thesis makes the following contributions:

1.) A new alpha-beta based algorithm (RAB) that simulates action executions in simultaneous move domains better than alternating move algorithms such as alpha-beta. The algorithm is a modification of an alpha-beta algorithm in which the order of player's moves is based on random numbers.

2.) An experimental analysis of several search and non-search based methods in our simultaneous move abstract combat domain. The methods are examined on the basis of quality of the solutions they produce, their real-time performance, and their suitability to different problem settings. The results show that the degree of interdependence between actions in a specific game set-up is the key for determining the methods that are most suitable in various situations.

3.) A number of methods (AB, RAB, and Monte Carlo) described in Chapter 4 that provide good moves in real-time in the abstract combat domain. Their effectiveness varies depending on the specific situation, but each method can produce reasonable moves under real-time constraints.

4.) A test environment for the abstract combat domain in which algorihtms can be examined by playing death-matches against one another. The environment has a generalized, object-oriented structure that allows designers to easily adapt it to other competitive domains (both simultaneous move and alternating move domains)

5.) A helper tool for creating balanced scenarios, which can be immediately used by game designers to create interesting scenarios.

Chapter 2 discusses the related literature. Chapter 3 describes in detail the problem domain used in this paper. Chapter 4 describes our methods to solve the problems presented. Chapter 5 evaluates different algorithms and Chapter 6 draws general conclusions on our findings. Finally, in Chapter 7 we present ideas for future research.

2. Related Work

Although real-time strategy games have become one of the most popular computer game genres over the last decade, the research in such games is in its infancy. Specifically, very few academic papers that are dealing with the general area of RTS gaming [8][10] and no papers related to abstract combat simulations were found in my literature survey. The military research community [15], is an area with ongoing research in the subject of combat simulations. In the military literature the area of predicting and simulating mutual attrition between two opponents is widely referred as combat attrition simulations. It is considered as one of the key aspects of modern combat modelling. Still, despite the fact that combat attrition is one of the most studied combat processes there is still no agreement on the best way to model it [15].

Most military researchers agree that understanding of combat phenomena is facilitated by using a hierarchy of combat to describe combat events and aggregate them for analysis [14]. At the top of such hierarchy is war, followed by a campaign, which refers to a sequence of battles closely associated in time and space that all share a single goal. A campaign is further subdivided into battles, which refer to a combat between two major forces. An engagement refers to a smaller scale battle between two forces. Finally, a duel is a fight between two individuals.

Since ancient times beginning 1500 years ago with Sun Tzu's The Art of War [38] a lot of efforts have been invested into developing fundamental laws and theories that help us understand the interactions of forces in the battles and predict the outcomes of such battles. In modern military combat models are widely used for battle planning, force sizing, human resource planning, logistics planning, and weapon purchase decisions [19].

In military research there are two main ways for unit representation in combat scenarios. First, are the non-aggregated combat models, where each unit is represented as a single entity on the battlefield. The second approach is aggregated combat models, where an

entity consists of several units. The study of aggregated combat models is the focus of most research in the area of combat attrition.

There are two types of aggregated combat models – homogeneous and heterogeneous [25]. In heterogeneous models, there is an interaction between aggregated units. Such models assess which unit attacks which enemy unit. In a homogeneous aggregated attrition process, all of the units are aggregated into a single unit. The interactions between different units are not considered in such models.

In modern military research there are two main ways for modelling combat attrition: Lanchester modelling and firepower score approach.

Frederick William Lanchester in 1914 formulated models for attrition. Their purpose was to justify the principle of concentration of forces under the conditions of "modern" warfare [12]. Lanchester then devised some models based on common differential equations to translate his hypotheses into mathematical terms. Currently, Lanchester-type attrition models refer to the set of differential equation models that describe changes over time in the force levels of fighters and other significant variables that describe the combat process [35]. Such models are used to answer such simple questions as who wins the battle or more complex questions addressing the force make up and tactics. Lanchester differential equation models have gained importance through their ability to provide insight into the dynamics of combat and their applicability [12]. While there is a wide variety of Lanchester-type differential models based on size and complexity, there are several factors that are common to most of them: attrition to a force is a function of force size and other associated attributes; force size is a function of time; for two opposing forces casualty rates can be written as a pair of differential equations; the solution to such models is a pair of functions giving sizes of both forces as a function of time (Refer to Section 2.1).

In the firepower score approach for combat attrition, models use force ratio in their structure. The ratio of attacker to defender combat power is used to determine the

casualties for both sides [25]. In this approach the combat power of a unit is computed by summing the combat power value for each particular weapon type. This is done both for attacking and defending forces. The next step is to divide the attacking forces combat power over that of defensive forces. This gives us a measure of a relative combat power in a battle. The issue of determining combat powers of units is a very difficult problem. There are several methods for computing firepower score values, based on military judgement and experience, such as RAND's ground force scoring system [1] and anti-potential-potential-method [25].

In military research, there are many studies that use both Lanchester models and firepower score approach. In [5][13][37] the authors did an empirical validation studes based on the data of WWII battles of Kursk and Ardennes. They have found that none of the Lanchester-type models fit the data perfectly, with Lanchester logarithmic model being the best predictor. There also are several studies that use the firepower score approach. In one such study [1], the value of a weapon system is varied as a function of the combat situation, dependent on type of terrain and on the type of battle. The ratio of attacking combat power to the defending combat power is defined by the environment both forces are operating in.

There are many other combat prediction models based both on Lanchester equations or on the firepower scoring approach have been widely used for analytical purpose to predict and simulate mutual attrition among two opponents (i.e. [22][5][1]). The main problem with such analytical methods arises from their exhaustive enumeration, which leads to exponential computation time. Therefore, such methods are not real-time and in practice can only be used to model small battles [41].

2.1 Lanchester Equations Example

Lanchester equations for modern warfare were developed on the principal that many units could concentrate fire on a single target. This was based on the fact that modern weapons

allow multiple units to engage a single opponent and can concentrate fire from weapons scattered on the battlefield [12]. He developed two types of differential equations: one for aimed scenarios, in which units have to aim at the opponent, and one for the area fire, where units do not aim at a specific opponent, but instead attack a specific area. We only show an example of Lanchester's aimed fire scenario (because such scenario is closely related to our abstract combat scenarios), where individual targets can be attacked by any number of opponents at the same time. Under aimed fire the attrition rate for force *X*, where *x* is the number of *X*'s units, depends on how many force *Y* units are shooting at it and the same holds for force *Y*, where *y* is the number *Y*'s units. The Lanchester equations for *x* and *y* are formulated as follows [12]:

$$\frac{dx}{dt} = -ay \qquad \frac{dy}{dt} = -bx$$

where, *a*, is the attrition rate coefficient for x expressed in terms of Xcasualties/(Yattackers * time) and likewise for b. From the above it can be shown that x and y are related by

$$b\left(x_o^2 - x^2\right) = a\left(y_o^2 - y^2\right)$$

where x_o and y_o are the initial numbers of forces for *X* and *Y*, respectively.

If $a = b$ (i.e. the attrition rates for forces *X* and *Y* are the same) and $x = 0$ (i.e. all x forces have been eliminated), the number of *Y*'s survivors are shown in $y_s = \sqrt{y_o^2 - x_o^2}$. Enumeration for some values of y_o shows the advantage of concentrating fire, which Lanchester aimed to illustrate originally. Specifically, as y_o (i.e. initial number of *Y*'s units) increases the number of *Y*'s casualties decreases significantly. Results are shown in Table 1.

Table 1: Shows the number of Y survivors and casualties as the number of initial y's units increases

y_o	Y's Survivors	Y's Casualties
100	0	100
200	173	27
400	387	13
800	794	6

3. Problem Description

Our primary objective when designing the experimental environment was to make it closely resemble fights in real-time strategy games. In addition, it is worthwhile to note that many RTS games were originally modelled after modern military forces. The most prominent examples are Rise of Nations by Microsoft Game Studios, Command and Conquer series by Westwood Studios and later EA Games, and the Warcraft series by Blizzard Entertainment. Therefore, our research is of interest to military researchers as well.

First, we examine a set-up of a typical real-time strategy game. Normally, a game will feature a number of unit types for each player type. This number can range from 2-3 unit types to dozens of units, depending on the game and also on the stage in a given game. Typically, when a player starts the game only a few basic units are available, however as the game progresses and the player gains access to more advanced technologies and gets more valuable resources the number of unit types they can produce can become very high. It is important to note that at any given time in a game, players will typically use only a few unit types out of an assortment of units available to them. Such restrictions can mostly be attributed to having more advanced technologies, whereby older units are not used since the player now has newer and more efficient alternatives to build.

Therefore, for our simulation we only need a few unit types that are representative of units present in a typical RTS game. We also have to remember that in *abstract combat* situations any unit can attack any other unit, thus making range and speed of movement irrelevant features. From examining the land units in some newer RTS games such as Rise of Nations, we can see that most land units can be divided into three classes as follows:

Strong Attackers / Strong Defenders / Average Shooters

- Such units have high attack power and very good defensive strength, but need some time to cool-off between attacks. An example of such a unit can be a tank.

Very Strong Attackers / Weak Defenders / Slow Shooters

- Such units have a very high attack power, but very weak armour and require long cooldown periods between attacks. A good example of such a unit can be artillery.

Average Attackers / Average Defenders / Fast Shooters

- Such units are average attackers and average defenders, but they do not need to cooldown between attacks. A good example of such a unit is a marine.

To summarize, each unit has three main attributes, which are attacking strength (attack value), defensive strength (hitpoints), and cooldown period. Another attribute is aiming time. When the simulation starts each unit picks its target and then both groups attack their targets simultaneously. The simulation continues until every unit in one or both teams is eliminated.

The above describes the experimental environment used for this research. The environment can be extended to be more realistic. However, even the simplified version of problem used here is quite challenging. In Section 3.1 we describe in detail the basic set-up of the problem, while Sections 3.2 and 3.3 describe two extensions to the basic

problems, which includes ranged units and units with defensive actions. In Section 3.4 we show an execution of two simple combat scenarios in our domain.

3.1 Basic Problem Set-Up

The simulation starts with two groups of units (3-4 in our experiments) facing each other. Since this is an abstract combat situation, all units have the ability to shoot at any other opponent unit, which makes unit speed and range irrelevant features in our situation. As outlined above we consider three main unit classes: *tanks*, *marines* and *artillery*. *Tanks* are strong offensively, strong defensively and have and average cooldown period. *Marines* are average offensively and defensively, but can shoot continuously. *Artillery* is very strong offensively, weak defensively and has a long cooldown period. Each unit has the following properties:

a. *Attack Value* – refers to the amount of damage a unit can inflict on an opponent at any given turn. For example, tanks and artillery will have higher attack values than marines.

b. *Hitpoints* – refers to the defensive strength of a given unit. For instance, a tank will have more hitpoints than artillery or a marine, and thus will be more difficult to eliminate.

c. *Cooldown period* - refers to how frequently a given unit can fire. A marine can fire continuously, while a tank needs to wait between shots for its weapon to be operable again. A tank with cooldown = 1, will shoot once then skip the next turn and during the third turn it can shoot again.

d. *Aiming* – before shooting, units have to select their target. In the next turn they can shoot *only* at that target. If a unit wants to shoot at another target it needs to re-aim. Re-aiming forces all units to skip one turn. During the

next turn the unit can shoot at a new target. Also, units with cooldown periods bigger than zero can use their turn to re-aim when their weapon is inactive. One exception to aiming occurs at the start of the simulation, when units can pick their target and shoot immediately. This is done to make the games faster, since the aiming penalty is the same for all units at the beginning of the game; it doesn't change the balance of power therefore it can be eliminated. Another exception is when a unit that is the current aim has been eliminated. Even if the unit that was aiming at the now eliminated unit has a cooldown period of zero (i.e. it can shoot), it has to skip the next turn to re-aim at another unit.

The games are played in our tournament environment, whereby each group executes its method and returns its move selection. Then both moves are executed simultaneously, by subtracting the attacker's attack value from the defender's hitpoints, by changing the current target, or by simply skipping the move as the unit cools down its weapon. Units die when their hitpoints are smaller or equal to zero after each round. Then the state of both teams is updated, and the simulation continues. The simulation is finished only when one or both teams are eliminated. All units have unlimited firepower, therefore a unit can not run out of ammunition. An example of such simulation is described in Section 3.4.

3.2 Defensive Actions Modification

In the basic problem set-up despite actions being executed simultaneously we theorize that the success of a player's action does not strongly depend on what the opponent chooses to do. We would like to test our algorithms in a set-up where there is much more interdependence between simultaneous moves. Specifically, in such situation the opponent to some extent can counteract any action of a player.

To test this hypothesis we have designed a simple modification of the original problem that makes the outcome highly dependent on the actions the opponent will take. With this modification knowing what the opponent will do in advance can be very advantageous to the other player.

The change is an introduction of a defensive action, which enables the player to use its attack value for defending rather than for attacking. Specifically, if a unit decides to defend instead of attacking, a certain proportion of its attack value (this proportion can be > 1) is added to its hitpoints. If this unit is attacked the attacker will only cause damage to the unit if its attack value is bigger than the unit's proportion of attack value used for defence. In addition, a defending unit can also cause damage to the attacker. The damage such unit can cause to the attacker is also proportional to its attack value.

A simple example demonstrates how a defensive action works. There are two opposing units as follows:

A (hitpoints: 10, attack value: 5)
B (hitpoints: 10, attack value: 6)

In this scenario the proportion of attack value for defence is 1.2 and the proportion to hit back the attacker is 0.2. If unit B decides to attack unit A and unit A decides to defend, unit B will not cause any damage since unit's A defence value is 5*1.2 = 6, which is equal to B's attack value. In turn unit A will hit back at B with 0.2*5 = 1 and this value will be subtracted from B's hitpoints. The new state will be:

A (hitpoints: 10, attack value: 5)
B (hitpoints: 9 , attack value: 6)

So, obviously there is a certain benefit to take a defensive stance, as demonstrated above. But there is also a certain degree of risk since a unit taking a defensive action might not be attacked and therefore it risks wasting its turn. The defensive modification makes the

outcome much more dependent on the opponent's actions. With defensive actions, knowing in advance what the opponent is planning to do can be of great benefit to the player.

3.3 Set-Up with Ranged Attacks

Although the basic problem set-up is an approximation to an abstract combat scenarios where units have unlimited ranges or are located very close to one another, in most combat situations the units are usually scattered on the battlefield and different units have different ranges. Therefore, it is too presumptuous to make an assumption that any unit can attack any other unit. Fortunately, the problem is easily expandable to accommodate for situations where not all opponent units are within range of every unit. The two new required parameters added to describe a unit are *weapon range* and *location*.

The addition of these parameters still does not make the scenario completely realistic since it does not take into account unit's movement and speed, which would make the simulation closely resemble reality. It does, however, agree with our approach of first tackling problems that are easier to solve and then gradually moving towards finding moves in more complex and realistic scenarios. Indeed, the situation with weapon range and location can be looked at as a snapshot of some units making certain moves. There can be many such snapshots considered. Then it would be possible to generate good moves for such snapshots and use the results in the general higher-level AI module.

The above can be a topic of future research. Currently we introduce the notion of *weapon range* and *spatial location* on a 2-dimensional grid. These problem extensions have the following properties:

1. *Location (x, y)* – Each unit has a spatial location on a 2-dimensional grid. The values for x and y are generated within certain predefined ranges. The area outlined by values of x and y is a playing field on which the units are located.

2. *Weapon Range* – Each unit has a range value for its weapon. *Weapon Range* determines how far from the current location a unit can fire. It can be described as a circular area with radius *Weapon Range* and centre at unit's location. Whether a unit can attack another unit is determined by their Euclidian distance.

3. *Connectivity* – represents the ability of units to attack each other. It is the number of units in total a team can attack divided by the total number of possible attacks when range and locations are not taken into consideration. This value is used for generation of scenarios with a certain connectivity and avoids scenarios that are either unconnected or have a very poor connectivity.

3.4 Abstract Combat Examples

We closely examine a typical set-up, where we have 2 groups with 4 units each. Both groups consist of two marines, a tank and an artillery unit. The example will consist of two parts, one that considers range and the other without range.

3.4.1 Unlimited Range Example

The attribute values for the three unit types are as follows (for the purposes of simplicity we use small values for unit attributes; the attribute values from actual simulations are different):

	Hitpoints (H)	Attack Value (A)	Cooldown Period (C)
Marine	2	1	0
Tank	4	2	1
Artillery	1	2	2

Team one consists of two marines, one tank and one artillery unit. Team two consists of three marines and a tank. Thus the initial set-up for the two teams is as follows (note: initially all units can shoot, including the ones with cooldown period bigger than zero):

	Unit 1	Unit 2	Unit 3	Unit 4
Team 1	H: 2 A: 1 C: 0/0	H: 2 A: 1 C: 0/0	H: 4 A: 2 C: 0/1	H: 1 A: 2 C: 0/2
Team 2	H: 2 A: 1 C: 0/0	H: 2 A: 1 C: 0/0	H: 1 A: 2 C: 0/2	H: 4 A: 2 C: 0/1

During the first move units pick their target and then shoot at that target. Let us assume the units pick the target as follows:

Team 1 (Unit:Target) (1:1, 2:1, 3:2 , 4:3) | Team 2 (Unit:Target) (1:1, 2:1, 3:3, 4:3)

Now the units execute their moves simultaneously. Unit 1 of team 1 attacks unit 1 of team 2, unit 2 of team 1 attacks unit 1 of team 2, unit 3 of team 1 attack unit 2 of team 2, etc. The resulting state after move execution is:

	Unit 1	Unit 2	Unit 3	Unit 4
Team 1	DEAD	H: 2 A: 1 C: 0/0 AIM: 1	DEAD	H: 1 A: 2 C: 0/2 AIM: 3
Team 2	DEAD	DEAD	DEAD	H: 4 A: 2 C: 1/1 AIM: 3

T1(team 1)(U1 (unit 1),U3 (unit 3)) and T2(U1,U2,U3) were eliminated. Also, note that the surviving units are now aiming at the opponent units the attacked. They can either shoot at the target they aim at or have to skip one turn to re-aim. For example, T1U2 can shoot because its cooldown period is 0 but it has to skip its turn since the target it has been aiming at was eliminated. Therefore it uses the next turn to re-aim. Also, cooldown period is now a factor. For example, T1U4 has to wait two turns before it can shoot since its cooldown period is 2. The next choice of actions and the resulting state for both teams is as follows:

Team 1 (2:Re-aim 4, 4:Re-aim 4) | Team 2 (4:Re-aim 4)

	Unit 2	Unit 4
Team 1	H: 2 A: 1 C: 0/0 AIM: 4	H: 1 A: 2 C: 1/2 AIM: 4
Team 2	DEAD	H: 4 A: 2 C: 0/1 AIM: 4

At this turn all units have to re-aim. All units of T1 are currently aiming at T2U4, since this is the only surviving T2 unit. Also, all units except T1U4 have their cooldown equal to zero so they can shoot. The next actions and state are as follows:

Team 1 (2:4, 4:skip) | Team 2 (4:4)

	Unit 2	Unit 4
Team 1	H: 2 A: 1 C: 0/0 AIM: 4	DEAD
Team 2	DEAD	H: 3 A: 2 C: 1/1 AIM: 4

Actions and state for the next turn are as follows:

Team 1 (3: 4) | Team 2 (4:Re-aim: 2)

	Unit 2	Unit 4
Team 1	H: 2 A: 1 C: 0/0 AIM: 4	DEAD
Team 2	DEAD	H: 2 A: 2 C: 0/1 AIM: 2

The final actions and state are as follows:

Team 1 (3: 4) | Team 2 (4:2)

	Unit 4
Team 2	H: 1 A: 2 C: 1/1 AIM: 2

Therefore, team 2 wins since team 1 has no units remaining.

3.4.2 Range Example

The same attribute values as in the first example are used, but there are additional attributes: location and weapon range. The teams are composed of one marine, a tank, and an artillery unit versus two marines and a tank. Although, in real simulations we calculate whether an opponent unit is within range of our unit, here for demonstration purposes each unit will have a set of units it can attack, called attack set (AS). The initial set up is as follows:

	Unit 1	Unit 2	Unit 3
Team 1	H: 2 A: 1 C: 0/0 AS: 1	H: 4 A: 2 C: 0/1 AS: 2,3	H: 1 A: 2 C: 0/2 AS:1,2,3
Team 2	H: 2 A: 1 C: 0/0 AS: 1	H: 2 A: 1 C: 0/0 AS: 2	H: 4 A: 2 C: 0/1 AS: 1, 3

Note that tanks and artillery have longer ranges, therefore their attack set has more units. The units actions and the resulting state are:

Team 1 (Unit:Target) (1:1, 2:2, 3:3) | Team 2 (Unit:Target) (1:1, 2:2, 3:3)

	Unit 1	Unit 2	Unit 3
Team 1	H: 1 A: 1 C: 0/0 AS: 1 AIM: 1	H: 1 A: 2 C: 1/1 AS: 2,3 AIM: 2	DEAD
Team 2	H: 1 A: 1 C: 0/0 AS: 1 AIM: 1	DEAD	H: 2 A: 2 C: 1/1 AS: 1, 3 AIM: 3

T1(U3) and T2(U2) have been eliminated. The next unit actions and the resulting state are:

Team 1 (Unit:Target) (1:1, 2:Re-aim: 3) | Team 2 (Unit:Target) (1:1, 3:skip)

	Unit 1	Unit 2	Unit 3
Team 1	DEAD	H: 1 A: 2 C: 0/1 AS: 2,3 AIM: 3	DEAD
Team 2	DEAD	DEAD	H: 2 A: 2 C: 0/1 AS: 1, 3 AIM:

But, already in this stage it is obvious that T1 has won since it can attack T2(U3), but T2(U3) cannot target T1(U2), since it does not belong to its attack set.

4. Abstract Combat Decision Algorithms

Before discussing the challenges in the abstract combat domain and the implemented algorithms we introduce several basic terms that are key to understanding this discussion. Those terms are mostly used to describe the search-based methods, however they are also used when talking about non-search based methods. When mentioning *search depth* we refer to a number of state transitions from the root of a search tree to the current state. For example, given a root state, with both players executing a single move results in search depth = 2. The *root node* is the state at which the search is started, while *leaf nodes* are nodes at which the pre-specified search depth was reached. Every leaf-node is assigned a

heuristic value by the *evaluation function*, which is an estimate on the value of that state. The nodes where the game has ended (i.e. win, loss, or draw situation) are called *terminal nodes*. All other nodes that are situated between the *root node* at the top and *terminal* and *leaf nodes* at the bottom are referred to as *interior nodes*. *Interior nodes* are nodes whose value is dependent on the value of its children. *Cut nodes* are nodes at which further search is unnecessary after exploring one or more children and therefore search can be stopped there.

The *branching factor* of any state refers to the number of successors that state has (or number of moves that can be executed at that state), while *move sorting* refers to the order of execution of those successors according to a certain heuristic criterion.

The structure of the chapter is as follows: Section 4.1 presents the motivation for developing our algorithms; Section 4.2 gives a detailed description of implemented methods; Section 4.3 describes the implementation of the evaluation functions; Finally, Section 4.4 talks about various improvements and applications.

4.1 Motivation

The major challenges in the abovementioned domain are the simultaneous move execution, huge branching factors and more importantly the limited time given to make a decision.

Linear programming (LP) provides a way for solving simultaneous move scenarios optimally. This requires a solution to a tree of LPs similarly to how it was done in the Oshi-Zumo game [9]. Such a computation is very expensive in real-time given currently available computation devices. Thus, other approaches need to be explored. For a detailed discussion about limitations of LP solutions in the abstract combat domain refer to Section 4.2.1.

Heuristic two-player search algorithms such as alpha-beta have been proven to work effectively in domains where the players alternate turns and where there is sufficient time to make a decision. However, search has not been applied to the more realistic scenarios in which players move simultaneously and have very little time to decide on what move to pick.

Indeed, applying an off-the-shelf version of an algorithm such as alpha-beta to our combat scenario will not be very useful, since its turn based execution of moves cannot be a good predictor of what is happening when moves are executed simultaneously. Therefore we propose a modification to the alpha-beta called *random alpha-beta (RAB),* which we believe simulates the simultaneous execution of moves well.

Another major issue is a large branching factor in our domain. This is because at any given turn all units in a group can attack any other opponent units at the same time. Thus, the number of possible moves generated at each turn equals to the number of all combinations of each of the units attacking each of the opponent's units. The branching factor is reduced somewhat by a *cooldown period and aiming,* whereby units have to skip one or more turns.

Even with such reductions, the branching factor remains very large. We propose two techniques to reduce it. One is using selective search, which concentrates on the moves that look more promising. The other is a smarter generation of moves and elimination of moves that are duplicates.

We have also created an alternate non-search-based algorithm, to test against our RAB search algorithm. The approach is called Monte Carlo, and it will be described in more detail later. First, however we will examine in detail the complexity of our domain. Then we will discuss the move dependence or independence in games with simultaneous moves. Follows is a discussion on the features of our domain that should enable us to use search methods effectively.

4.1.1 Examination of Problem Complexity

In a game of chess only one piece can move in any given turn. Therefore, the branching factor is calculated by counting all of the moves at the given turn. Even though some pieces can have many different options to move at any given time, there is still an upper bound on the total number of moves possible in any position in the game of chess. More precisely: (# moves < # pieces * board size)

Our game is quite different since at any given time all of our units can act at the same time. Therefore, the branching factor is exponential, since every time a unit is added, the previous number of possible moves is multiplied by the number of actions that a new unit can execute. There is still a ceiling on the number of actions possible given the number of units involved, but we do not have a restricted board size like in most board games. Ideally, we would like to solve for instances with any numbers of units (i.e. 0...n). For example, let us examine a game of 4 units versus 4 units. In the initial position, there is 4^4 = 256 possible attack actions. In a game of 5 units versus 5 units, the branching factor increases to 5^5 = 3125. This can be continued, however, the trend can be clearly seen already.

Luckily, there are other factors specific to our domain that help to make sure the branching factor will be reduced. Cooldown period and aiming ensure that the branching factor decreases in deeper tree positions. Also, "natural causes" (i.e. units being killed) reduce the branching factor as the game progresses. Unit elimination in our game happens much faster than in a typical board game. Specifically, in the non-defensive scenario every action results in a significant reduction of opponent's hitpoints and vice versa. Eventually, there will be fewer units, which leads to a considerably smaller branching factor. Still, the branching factor is the key obstacle for creating effective algorithms to solve our various combat scenarios.

4.1.2 Dependence in Games with Simultaneous Moves

Another feature of our domain that does not occur in games like chess or checkers is simultaneous move execution, which makes the issues present in our domain similar to problems researched in the field of game theory in economics and mathematics. The famous prisoner's dilemma [26] and Rock-Paper-Scissors games come to mind. In such games the result of an action of a player depends on what the second player chooses to do. This makes our domain an incomplete information domain, where the success of player's next action will depend on the move its opponent chooses.

Linear programming (LP) provides a way for solving simultaneous move scenarios optimally [9]. However, considering our branching factor, solving this problem, particularly solving it in real time, using LPs is currently infeasible. For detailed discussion about limitations of LPs in the combat domain refer to Section 4.2.1.

Fortunately, the dependency on opponent moves is not a black and white subject. That is, there are simultaneous move problem domains where there is a high correlation between the success of a certain action and the opponent's action, while there are also situations where such correlation is very small or even non-existent. For example, a game of Rock-Paper-Scissors is a domain where there is a perfect correlation, that is, no matter what your action is, the opponent action can always determine the final outcome of the game (i.e. draw, win, or loss).

Our abstract combat domain consists of two scenarios: the defensive scenarios and the non-defensive scenarios. In the non-defensive scenarios, we predict that there is relatively weak interdependence between player's actions. Since when units attack their opponents, no matter what the actions of opponents will be, those units will still cause damage to the opponents and the opponents will end up with reduced number of hitpoints. In the defensive scenarios, the success of an action depends more on what the opponent's action will be. For example, if a unit attacks an opponent unit that chooses to defend itself, its attacking action will likely not be successful. But if the opponent unit chooses to perform

an attacking action, the outcome of attacking that unit will be successful. Therefore, we expect that in non-defensive scenarios a simple deterministic method might prove almost as effective as the RAB algorithm, while in defensive scenarios the success of RAB should be more noticeable. This allows us to hope that our randomized search based method can be applied effectively to find good moves in simultaneous move domains.

As far as move dependency goes, there is a relatively weak dependency between the players' moves when defensive actions are not allowed. That is, there is not as much advantage for one player to know, what the opponent's move selection will be in advance in a single turn, as in the defensive scenarios. We intend to show how our methods perform as the degree of dependence in the scenarios changes and give specific examples where such knowledge would be of some benefit to a player. One of our goals is to see how a randomized approach performs compared to a deterministic approach as the degree of dependence changes.

4.1.3 Tackling The Challenges of the Domain

From the above we know that the branching factor and the move dependency arising from the simultaneous move executions are the main challenges when looking for ways to solve our problems. Knowing the challenges is only the first step to trying to understand the domain. The second step is trying to find the features in the domain that will enable us to overcome such challenges and help us design effective real-time algorithms.

The only way of tackling the huge branching factor both for search based and non-search based (i.e. Monte Carlo approach, linear program, random) approaches is by reducing the number of moves that are considered at every state in the game. In our domain there are several ways to achieve this. One approach is trying to remove moves that are duplicates (i.e. moves that are symmetric to each other). Another approach is to only consider a small subset of promising moves at every state of the simulation.

Removing duplicate moves from being considered is more of an engineering problem than a research one. In a typical scenario there are units that have identical hitpoints, identical attack values and are in an identical state at a given point of the simulation. Thus, actions by each of the units against a specific opponent are identical to each other. Therefore, it is feasible to just consider one of such actions. The technique to do this will be described in detail in Section 4.4.2.

The key to reducing the branching factor is by considering a very small subset of promising moves. Finding such moves requires having an effective evaluation function at interior nodes, which can predict the "goodness" of a current position, meaning that the evaluation function should be able to predict the balance of power between opponents with high degree of accuracy. We hope that in our domain (i.e. both in defensive and non-defensive settings) the evaluation function would be able to successfully find a subset of promising moves on which our methods can concentrate their additional search effort to further narrow down the selection.

4.2 Implemented Algorithms

4.2.1 Linear Programming

The Minimax theorem was developed and proven by John von Neumman. It states that in every *zero-sum* two player games there exist optimal *mixed strategies*. The term zero-sum refers to games where a gain for one player is an equal loss to the opponent. Mixed-strategies refer to the probabilities assigned to moves in different states of the game. The moves are picked according to these probabilities. The games in abstract combat domain are zero-sum games that have optimal mixed strategy solutions.

John Nash proved that not knowing the actions of the opponent forces us to consider mixed strategies and that Nash-optimal mixed strategies exist for any matrix game [24].

Nash-optimal strategies for simultaneous move games can be found by solving linear programs (LPs).

To find a Nash-optimal strategy for a given state in our abstract combat game we need to solve two linear programs one for the MAX player and the other for the MIN player. Figure 1 [9] shows the two LPs that need to be solved. The pseudo-code for the program that uses the LP solver from Oshi-Zumo game [9] is shown in Figure 2.

Maximize Z such that	Minimize Z such that
For all $1 \le j \le n_2 : Z \le \sum_{i=1}^{n_1} A_{i,j} x_i$,	For all $1 \le i \le n_1 : Z \ge \sum_{j=1}^{n_2} A_{i,j} y_j$,
For all $1 \le i \le n_1 : x_i \ge 0$,	For all $1 \le j \le n_2 : y_j \ge 0$,
$\sum_{i=1}^{n_1} x_i = 1$	$\sum_{j=1}^{n_2} y_j = 1$

Figure 1: Shows the LPs for two players for computing Nash-optimal strategies. Player "Max" has moves from 1... n_1 , while "Min" is from 1... n_2 . $A_{i,j}$ is the score from "Max"'s perspective if a pair of moves (i, j) is chosen. x_i and y_j are probabilities of choosing moves i and j, for "Max" and "Min", respectively.

A solution of each of the linear programs is the probability distribution for all of the moves available for one player. The player then selects the move according to the move's probability. For instance, in Rock-Paper-Scissors all moves have exactly the same probability of ⅓ in the Nash equilibrium. Therefore, a Nash-optimal solution produced by solving the associated LP will not win anything against even the simplest strategy, such as playing Paper all of the time. However, in more complex games such as poker or our abstract combat domain, some of the moves are assigned a zero probability and therefore the Nash-optimal player will never select such moves. Moves that are assigned a zero probability may lead to a loss situation if the opponent is a Nash-optimal player. Thus, other strategies that select such moves occasionally can lose against the LP player in the long term.

```
Move LPPlayer (State state){

  Moves movesToMove [1..movesToMove.size()];
  Moves movesOpponent [1..movesOpponent.size()];

  GenerateMoves (movesToMove);
  GenerateMoves (movesOpponent);

  int n = movesToMove.size();int m = movesOpponent.size();

  //executes all combinations of (player, opp) moves and stores scores of each pair in input
  crossProductOfMoves(n, m, movesToMove, movesOpponent, input);

  //LP solver; its output is a probability distribution stored in output array
  MinMaxLp (n, m, input, output);
  double randProb = random number in [0,1];

  double totalProb = 0;

  for i=1..n {

     totalProb += output[i];
     if (randProb < totalProb){
        bestMove = movesToMove[i];
        break;
     }
  }
  return bestMove;
}
```

Figure 2: Pseudo-code for the LP player's algorithm

LPs provide a way of finding optimal solutions for our abstract combat domain. Their biggest problem however is the inability to solve problems in real-time. The computation required for solving LPs grows as a proportion of the branching factor as the depth of look-ahead increases. For example, when performing a depth = 2 search with 100 actions for each player, the LP needs to solve a 100x100 matrix, which is manageable. Increasing the depth to 4, even with decreasing branching factor of 50, requires an LP solution for a 50x50 matrix for each entry in the original 100x100 matrix and then populating that matrix (100x100) with expected results from the respective 50x50 solutions. Finally, the LP solver needs to compute an optimal strategy (i.e. probability distribution for all the moves) for the populated 100x100 matrix. Such computation is quite costly and it is already impossible to solve in real-time in our domain. Thus, the LP method for our domain cannot comply with real-time constrains and can only perform limited depth

searches (specifically depth 2). However, in simultaneous move domains search depth was shown to be not as essential as in alternating move domains. Such results were shown in a game of backgammon [17]. It will be interesting to see whether such results hold true for our abstract combat domain.

4.2.2 Alpha-Beta (AB)

Undoubtedly, heuristic search has had the most influence on the game playing programs in artificial intelligence. The most important of the heuristic search algorithms to which most advances in two-player games can be attributed is the alpha-beta algorithm [6]. Our research also relies heavily on the alpha-beta algorithm (αβ). The base case search-based player is implemented using the standard version of the alpha-beta algorithm. αβ is an enhancement to the min-max algorithm for two player games. In min-max type algorithms each of the players is trying to maximize their results and minimize the result of their opponent.

The alpha-beta algorithm maintains two bounds alpha (α) and beta (β). Alpha is a lower bound of the best score that the player to move can attain, while beta is an upper bound on what the adversary can attain. When $\alpha \geq \beta$ the search is stopped at that node, since a maximizing opponent can achieve a better minimum score elsewhere. In our work we use a Nega-Max [21] formulation of the αβ algorithm, since it is somewhat easier to implement than the min-max version of αβ because we can only maintain one bound (MAX). In the Nega-Max [21] version when going down the search tree, the α bound becomes $-\beta$ (i.e. $\alpha=-\beta$) and β bound becomes $-\alpha$ (i.e. $\beta=-\alpha$), but when going up the tree the returned score is negated. The Nega-Max [21] modification ensures that the α and β bounds as the well as the returned score are from the perspective of the player executing the move. For example, the α bound, is the minimum score player one can achieve already. Thus, the $-\alpha$ is actually an upper limit (i.e. β) on what the second player can achieve.

4.2.3 Random Alpha-Beta (RAB)

Our goal when designing the RAB algorithm was to overcome the major disadvantage of the basic alpha-beta algorithm of not addressing possible move dependencies in domains with simultaneous move executions. Alpha-beta is an alternating move algorithm therefore it gives each player an advantage of knowing the other player's move. This makes alpha-beta a somewhat inaccurate tool for simulating the execution of actions in our abstract combat domain, since none of the players know in advance what the other's move will be.

To soften the effect of advance knowledge of opponent's moves in a game we propose an approach where the order of move execution in pair-wise horizontal layers through the search tree is randomized. Thus, at some nodes in a tree player one will have an advantage of knowing player two's move, while at other nodes the situation will be reversed.

The algorithm works as follows. First, it divides the alpha-beta tree into horizontal layers of depth two spanning the entire tree, from top to bottom. At each node at the top-level of a given layer the choice of which player is to move is randomized. Thus, at each such node there is an equal chance for either player one or player two to move. The only exception occurs at the root of the tree where the first move always belongs to the player performing the search and its moves are always followed by the opponent moves. Children of the root node are not randomized and are assigned the opposite colour to that of the root node. A sample RAB tree is shown in Figure 3 and the high-level algorithm pseudo-code for RAB is presented at Figure 5. Figure 4 shows two binary minimax trees instead of the alpha-beta tree. Both the randomized and the alternating minimax trees have the same structure, number of nodes and values at the leaf nodes. The difference is in the order of move execution and in the value propagated to the root.

The aim of RAB is to more closely approximate what happens when moves are executed simultaneously, since randomization of move order is a more natural simulation for

simultaneous move execution. Specifically, with RAB our main hope is that the advantage to a player will be minimized, since in the whole search tree both players will get roughly equal chances of knowing the opponent's moves.

For the abstract combat domain, however, there is a need for the algorithm to perform under tight time constraints. RAB, as opposed to the base version of the alpha-beta algorithm is not deterministic, because every run of the algorithm is likely to produce different results. Being a statistical approach it requires multiple runs to be performed for every move, which is available to the player executing the search at the root. For every such move, the best scores will be recorded for every run. Then for each move the average score and the standard deviation are calculated. Multiple runs are required, because when dealing with randomized algorithms performing a single run or very few runs is not sufficient for drawing valid conclusions about the quality of a given method. The move that is chosen for execution is determined by taking the average scores and standard deviation of scores of all runs for each considered move. The move with the best combination of average score and standard deviation is then executed. The combination of average and standard deviation to be used is determined experimentally.

In each run, the order of players' moves will be randomized producing different scores for the player. We think that the average and standard deviation for each move over all the runs, better simulates move execution in simultaneous move domains, than a single run of a regular alpha-beta algorithm. However, the main concern with RAB is on the number of runs that it will require to find the best or close to the best move. Also, if there is very little advantage of knowing the opponent's moves in our domain, it is possible that alpha-beta can find a good move or possibly the best move in just a single run. It is also possible that it is more worthwhile to invest the extra resources into deeper searches rather than on repeated searches.

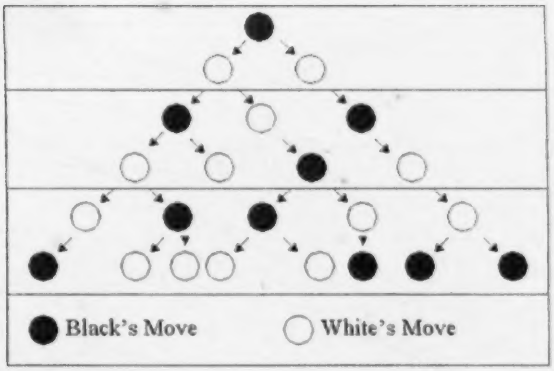


Figure 3: Shows a sample RAB tree. The black player moves first at the root followed by the white player. At the next level, the player to move is randomly selected..This player's move is followed by the move of the opponent of the randomly selected player.

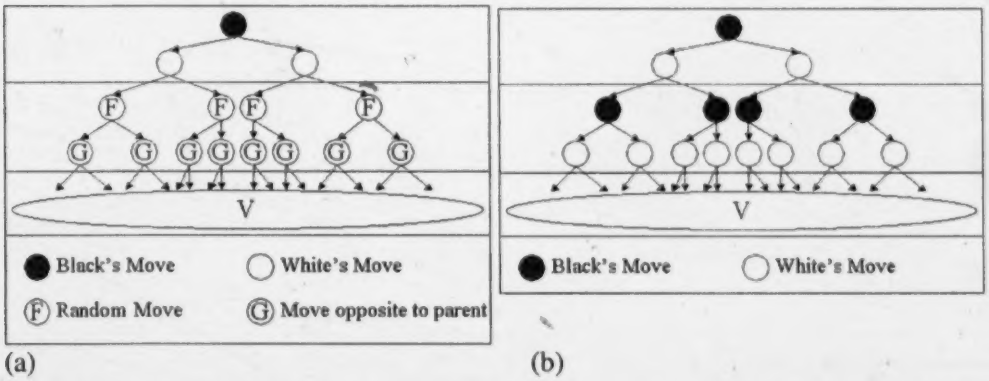


Figure 4: Randomized binary minimax tree (a) and alternating binary minimax tree (b). Both trees have the same structure, number of nodes and exactly the same values at leaf nodes. The difference is in the order at which the moves are executed and the value that is propagated to the root.

```
Move GetMove (State state){
    vector  of vector of double  allMoveScores;
    int i = 1; depth = 2;

    while (current_node_count <= node_count_limit){
       i++;
       if (enoughRuns)
          depth += 2;
       TopLevelRAB (state, allMoveScores[i], depth);
    }
    return bestMove = GetBestMoveScore (allMoveScores);
}
////////////////////////////////////////////////////////////////////////////////
TopLevelRAB (State state, vector of double  moveScores, int depth){

    Move moves[1..moves.size()];
    GenerateMoves(moves);

    //goes through a single run of all generated moves
    for i = 1..moves.size() {
        newState = makeMove (moves[i]);
        score = RAB (newState, alpha, beta, depth, randGenerate = 0);
        moveScores[i].score.append(score);
    }
}
////////////////////////////////////////////////////////////////////////////////
RAB (State state, int alpha, int beta, int depth, int randGenerate){

    if (terminalNode || depth == 0) return evaluate (state);

    score = -∞;

    if (randGenerate){
        playerToMove = (int)random number in [0,1];//randomly select either 0 or 1
        randGenerate = 0;
    }else{
        toMove = opponent; randGenerate = 1;
    }
    if (parentPlayerToMove == currentPlayerToMove){
        alpha = -beta;
        beta = - alpha;
    }
    GenerateMoves (moves);
    for i = 1..moves.size() {

        newState=MakeMove(moves[i]);
        value= -RAB (newState,-beta,-alpha, depth-1, randGenerate);
        if (value > score) score = value;
        if (score > alpha) alpha = score;
        if (score >= beta) break;
    }
    if (parentPlayerToMove == currentPlayerToMove) return –score;
    else return score;
}
```

Figure 5: Pseudo-code for the RAB algorithm

RAB Implementation and Score Calculation Details

The RAB algorithm is implemented using an iterative deepening [32]. Iterative deepening is often used in environments with real-time constraints. It performs multiple searches starting with the lowest depth and increases the search depth at every successive iteration. The rationale behind the technique is that lower depth searches take only a fraction of what the next higher depth search will take and thus there will be only a minimal waste of resources. The main benefits are that at anytime a reasonable solution is available and a guarantee that a search algorithm will complete in real-time, with a good likelihood that the highest depth will be reached given the resources available.

In our case the initial depth for iterative deepening [32] search is two and at every successive search the depth is increased by two. Only even depths and depth increments are used because in simultaneous move domains both players execute their actions at the same time. An odd depth will make the search biased towards the player who starts the search, since that player will have one more move than the opponent (such a situation is impossible in simultaneous move games).

The complication when implementing iterative deepening with RAB arises due to RAB's need to complete several trials at a single depth. The higher the search depth the more iterations random alpha-beta needs to complete at that depth. This is because the deeper the search the higher the variability or standard deviation of the results. Therefore, more RAB iterations are required to get a high accuracy prediction.

The formula we use to calculate the number of runs at each depth is $\left(\sqrt{2}\right)^{(2+depth)}$. This ensures that at each successive search depth, because of higher variability in results, the RAB will be able to complete exponentially more runs than at the previous depth. Also, in contrast to a simple version of iterative deepening, where only the results from the last depth search are used, in our version we accumulate the results of all iterations starting with depth two and finishing with the last completed search. Unlike searches in traditional two-player domains where increasing the search depth usually results in better

accuracy and the results of previous iterations can be ignored, in a simultaneous move domain there is a dependence on what the opponent will do. Therefore, results from lower search depths should not be ignored. It is possible that even a lower depth search can produce a very good result (which can determine the winner) depending on what the opponent does.

4.2.4 Monte Carlo

Monte Carlo methods solve problems by executing a large number of random or pseudo-random actions and examining the numerical results such actions produce. The method is used for finding solutions to problems that are too complex to solve analytically. Stan Ulam first developed the 'Monte Carlo method' [39]. His original method was developed to acquire solutions to complex mathematical problems, using a statistical sampling method with random numbers. Monte Carlo [39] methods are used in many areas. In computing science, Monte Carlo methods have been used successfully to find solutions to incomplete information problems or problems with large state spaces, specifically in the game of Go [3]. Of course the way the methods are applied varies widely from field to field, and there are many variants of Monte Carlo methods.

In our application we use an approach similar to the one that proved to be successful in the Olga Go program [3]. The Olga Go program has very little knowledge of the actual Go domain, but it was shown to perform on par with the previous knowledge-based system. In our approach we play out a game until one player is eliminated. For each of the main player's (i.e. the player who performs the simulation) moves at the root, a series of simulations is performed given the available resources. After the runs, the average scores and standard deviations for each move at the top are computed. The move with the "best" average score and standard deviation combination is selected by the player to be executed. Such combination is determined experimentally.

At every turn the player randomly selects from the available actions and then executes the selected action. The run continues in this fashion with both players executing their randomly selected moves, until one of the players is eliminated. At that time the attributes of the units of the surviving player are used in the calculation of one of the evaluation functions (refer to Section 4.3). The value is then propagated to the top node and is recorded as one of the values for the selected move. For the pseudo-code for the Monte Carlo method refer to Figure 6.

```
Move GetMove (State state){

   vector of vector of double allMoveScores;
   int i = 1;

   while (current_node_count <= node_count_limit){ //quit if  #nodes exceed limit
      i++;
      allMoveScores.resize(i);
      TopLevelMonteCarlo (state, allMoveScores[i]);
   }
   return bestMove = GetBestMoveScore (allMoveScores);
}
////////////////////////////////////////////////////////////////////////////////
TopLevelMonteCarlo (State state, vector of double  moveScores){
   Move moves[1..moves.size];
   GenerateMoves(moves);

   //goes through a single run of all generated moves
   for i = 1..moves.size() {
        newState = MakeMove (moves[i]);
        score =  MonteCarlo(newState);
        moveScores[i].append(score);
   }
}
////////////////////////////////////////////////////////////////////////////////
MonteCarlo (State state){

   Move moves[1..moves.size()];
   GenerateMoves(moves);
   int moveChoice = random of [1..moves.size()];
   newState=MakeMove(moves[moveChoice]);

    if (player1NumUnits == 0) || (player2NumUnits == 0) //guarantees termination
       return Evaluate (state);
    else
       MonteCarlo (newState);
}
```

Figure 6: High-level pseudo–code of the Monte Carlo method

Our hope in using the Monte Carlo simulation is to see whether by repeatedly pseudo-randomly selecting moves we will be able to find good moves. However, in a real-time domain the key is not just finding the best or close to the best move but finding such a move given very limited resources. What will determine its suitability for a domain with real-time constraints is the quality of solutions the Monte-Carlo method can produce given strict-time constraints.

One likely advantage of Monte Carlo simulation over alpha-beta and RAB in a domain with move dependencies and simultaneous move executions, is the fact that Monte Carlo incorporates move dependencies in its structure, while the search-based methods do not. In a Monte Carlo simulation each player selects its move randomly out of its successor move set. Thus, an action a given player selects in no way depends on the action its opponent chooses.

4.2.5 Random Player

The random player is the simplest base case method in our abstract combat domain. It works as follows: at every turn all actions are generated, then both players randomly select one of the actions and finally execute them. The random player was created for the purpose of testing and simple experiments. That is, if our other methods did not beat the random player by a large margin, there is a bug or they should not be even considered. The only likely benefit of a random player is its speed, since it requires very few resources to produce a "solution".

4.3 Evaluation Functions

Before describing in detail the specifics of our domain and the factors behind choosing specific evaluation functions, we examine desired guidelines most evaluation functions should adhere to. The key attributes for any evaluation function are:

1. *Accuracy* – how accurate is the evaluation function in correctly assessing the value (i.e. balance of power between the opponents) of the game state
2. *Speed* – the speed of computation is important, because an evaluation occurs at the bottom level of the tree and complex evaluation functions can significantly hinder the performance of search. It is desirable to have an evaluation function that is very accurate in assessing the game, but is still fast to compute.

For evaluation functions it is crucial regardless of the domain to be able to predict the goodness of any non-terminal game state with a high degree of accuracy, since most searches cannot reach terminal positions. This is especially true when considering the real-time constraint of the abstract combat domain. Another issue to consider when designing the evaluation function is its speed. Speed is important because the evaluation occurs at the leaf nodes of the tree, where there is the largest number of nodes. In general, due to the branching factor the performance of any search is proportional to the execution at the lowest layer. For example, an evaluation function that is two times faster than a more complex evaluation function will make the overall search performance to be almost twice as fast. Therefore, there is often a trade-off between speed and accuracy when designing an evaluation function.

Next we present the evaluation functions that we designed for the abstract combat domain. We start with the simplest evaluation in 4.3.1 and continue with square root evaluation in 4.3.2. Finally, we describe the most complex and computation intensive Tournament Evaluation function in 4.3.3.

4.3.1 Simple Evaluation Function

The simple evaluation function takes into account all of the attributes of each unit: hitpoints, attack value, and cooldown period. The function has the following form:

$$\sum_{i=1}^{n_1} \frac{h_i^{(1)} \times a_i^{(1)}}{c_i^{(1)} + 1} - \sum_{i=1}^{n_2} \frac{h_i^{(2)} \times a_i^{(2)}}{c_i^{(2)} + 1}$$

where $h_i^{(j)}$ = hitpoints, $a_i^{(j)}$ = attack value, and $c_i^{(j)}$ = cooldown of unit i of player j

The idea behind the design of the Simple evaluation function is to estimate the lifetime damage units can inflict. In general, hitpoints (h_i) estimate the life expectancy of a given unit, while attack value over cooldown plus 1 (i.e. $a_i/(c_i+1)$) represents the average damage a unit will deal during each time unit.

One of the weaknesses of this evaluation function is its inability to capture the difference in cooldown periods. For example, if there are two units, one with attack value = 10 and cooldown = 1 and the other with attack value = 5 and cooldown = 0, by computing the *(Attack value / (cooldown period+1))* ratio in our formula we get the same result. However, it is obvious that the unit with zero cooldown period is a better unit, since it can deal the damage earlier and more often. Thus, the main advantage of a unit with a smaller cooldown period is its flexibility. It can change targets more often and kill off opponent units that have few remaining hitpoints earlier and with less waste of its attack points than a unit with a higher cooldown period.

Another weakness of the Simple evaluation function is its inability to distinguish between hitpoint distributions of units. In general, it is more beneficial for a player to have units that have more uniform hitpoint distributions than having some units with very low hitpoint values, while other units with very high hitpoint values. In most cases, units with very low hitpoints values are much closer to elimination than units with higher hitpoints values. Thus, in groups that are composed of some units having low hitpoints and other

units having high hitpoints, the low hitpoint units will be eliminated relatively fast, meaning that only high hitpoint units will remain to attack the opponent. In scenarios where hitpoint distributions are uniform, the expected lifetime of all units in which they can cause damage to the opponent is longer than that of units with lower hitpoints. The simple evaluation function only looks at the overall number of hitpoints in a team. Therefore, if there are two teams with the same number of hitpoints, one with a uniform distribution, while the other having a non-uniform distribution, everything else being equal, the result of a simple evaluation function will be the same.

Thus, its two main drawbacks are its inability to explicitly emphasize eliminating units and its failure to take the difference in cool-down periods into account.

4.3.2 Square Root Evaluation Function

To address the hitpoints distribution problem of the simple evaluation function we propose a straightforward modification, which is done by taking the square root of the hitpoints of each unit. By applying the square root to hitpoints the evaluation function implicitly prefers having units with more uniform distribution of hitpoints to ones with a more uneven distribution. The new square root evaluation function is as follows:

$$\sum_{i=1}^{n_1} \frac{\sqrt{h_i^{(1)}} \times a_i^{(1)}}{c_i^{(1)} + 1} - \sum_{i=1}^{n_2} \frac{\sqrt{h_i^{(2)}} \times a_i^{(2)}}{c_i^{(2)} + 1}$$

where $h_i^{(j)}$ = hitpoints, $a_i^{(j)}$ = attack value, and $c_i^{(j)}$ = cooldown of unit i of player j

To illustrate the effect of the square root, let us examine a simple situation. Consider two groups with equal numbers of hitpoints as follows:

Group A (10, 3, 17)

$\sqrt{10} + \sqrt{3} + \sqrt{17} = 9.02$

Group B (10, 12, 8)

$\sqrt{10} + \sqrt{12} + \sqrt{8} = 9.45$

From the above example we can see that the more uniformly distributed set (i.e. group B) is assigned a higher value. Thus, a more uniformly distributed hitpoint values will be preferred. Another effect of taking the square root of the hitpoints is more emphasis on the unit attack values. There is no clear intuition on how increasing the importance of attack values will affect the performance; therefore this will be answered in the experiments. The performance of the Square Root evaluation function is proportional to the number of surviving units for both teams. However, we have to note that the Square Root evaluation function still does not address the cooldown problem described above. The tournament evaluation described next will try to fix that problem.

4.3.3 Tournament Evaluation Function

The final evaluation function that we will be using in our tests is the Tournament evaluation function. The main goal of a Tournament evaluation is to overcome the shortsightedness of both Simple and Square Root evaluation functions. Because the depth of the search rarely reaches the end of the game there is a need to come up with a better estimation of the end game positions at the leaf nodes.

At each leaf node the Tournament evaluation function plays out a simplified game until one player's units are eliminated. The games are a simplified versions of our simulation, where all attackers' *attack values* are summed, then divided by the number of the defenders, and subtracted from each defender's *hitpoints*. At the end of such a game only one side is remaining. Its units are evaluated to calculate the score using either the Simple or the Square Root evaluation. This evaluation function is more computation intensive, however, it should produce more accurate prediction of the future.

The tournament evaluation takes cooldown periods into account in its simulations, as attack values of units that have to skip a turn are not counted at that turn. The only

property that the Tournament evaluation function does not consider is aiming, since for simplification purposes it is assumed that attack values are distributed equally between the defenders. Obviously, the Tournament evaluation function is a simplified version of our combat game and it will not always produce correct predictions, but it provides an explicit way to look at the possible future state. Its other major weakness is its performance, since unlike the other two simpler evaluation functions, whose performance is proportional to the number of units remaining in the game, the Tournament evaluation function's performance is proportional to the number of units multiplied by the length of the simulated game.

4.4 Other Improvements and Applications

4.4.1 Selective Search

There are two main ways to combat the large branching factor we encounter in our domain to enable algorithms to perform effectively in real-time. The naïve solution would be to decrease the depth of the search, so even with large branching factors the algorithm will be able to come up with solutions in real-time. However, as we know from other domains, decreasing the search depth leads to lower quality of the result [].

However, in some domains (eg. Backgammon [17]), a limited depth search will produce a subset of moves that is likely to contain the best moves available.

Therefore, before the start of each of the proposed methods at each level in the tree we perform a complete depth one search for each of the successors. After that, the top N successors are sorted in the decreasing order of their scores. Then the search will concentrate only on those top N successors. Because of the decreased branching factor the search can go much deeper. For such a sorting to be successful, it is important to have an evaluation function that can provide an accurate estimate of the state value.

4.4.2 Removing Duplicates

Another way to decrease the branching factor is by avoiding the generation of all the moves. We can get rid of moves that are duplicates. In particular, a small group of units consists of units of 3 types: *marines*, *tanks*, and *artillery*. Usually there are 2 or more identical units of some type present in a group. Let us consider a situation where there are 2 or more units that are identical (i.e. they have the same attack value, hitpoints, current cooldown period, and aim). Generating all possible combinations of such units attacking the opponent is redundant.

For example, if we have 3 identical units and there is one opponent unit *i* that one of these units wants to target. It is sufficient to generate only one action for one of our units to target *i*, since it does not matter which particular unit targets *i*. Thus there is a saving of order of three in the actions generated in this example. We can refer to such units as interchangeable units. In general, we only need to consider all unique combinations, for each of the group of interchangeable units targeting the opponent.

Permuting out such repeated moves would significantly reduce the number of successor moves. Our move generation routine generates only unique combinations of such moves for every group of interchangeable units.

4.4.3 Delayed Move Execution

Delayed move execution is required when using turn-based algorithms such as alpha-beta in domains with simultaneous move executions. In alternating move games, the first player will change the state of the game by executing its move. After that the second player will change the state of the game by executing its move. Such a sequence of events does not represent our situation accurately, since the state of the game should have remained the same when the second player decides on its move.

Therefore, in the abstract combat domain we should have a situation where player one decides on its move. Then without changing the state of the game, player one passes its move to the second player. In turn the second player decides on its move based on the unchanged state of the game and finally both players execute the moves simultaneously. This avoids the incorrect simulation of the game where after the first player executes its move the state changes in such a way that actions, which should have been available to the second player, are not present anymore. For example, without delayed move execution, when player one eliminates one or more of opponent's units with its move, these eliminated units will not be able to cause damage to player one's units.

4.4.4 Balancing Scenarios Application

In most RTS games there is a set of custom designed sub-games, called skirmish scenarios. In such scenarios players cannot build units and infrastructure; their only goal is to achieve military domination by eliminating all other players. The main issue is how to design such scenarios, so that each player has an equal chance to win (i.e. their combined unit strength has to be the same). The naïve solution would be to assign each of the players exactly the same numbers and types of units. Such solution, however, would not make a very interesting scenario, since having the same force makeup would lead the players to use exactly the same strategy. Moreover, it reduces the replay value of the scenario, since human players would like to play the same scenario again using different starting positions (i.e. different player "nationality" or type).

Scenario designers usually know what the approximate make-up of each team in a scenario should be. For example, let us assume they would like to design one team that consists of 3 marines and one tank and the other team that consists of 2 tanks and one artillery unit. Then, how do they decide whether the teams are of equal strengths? Currently, they have to perturb each of the parameters manually, give the scenario to the testers to play, and wait for their comments. This cycle can continue until designers are

satisfied with the scenario. This can be a tedious and labour intensive job, which, despite of all the effort invested, often produces unbalanced scenarios.

Instead, it is possible, with minimal modification to use one of our proposed methods (i.e. which ever algorithm turns out to be the best), to balance the units automatically. When using our application the designers provide the initial mark up of both teams, specifying the type and attributes (i.e. hitpoints, attack value, and cooldown period) of each of the units in both teams as an input. They also specify the team whose attribute values they would like to change.

The simulation starts with both teams playing out the abstract combat scenario. Depending on the result for the team whose attribute values we would like to change, one of the following happens:

1. WIN – if the team whose attribute values are to be perturbed wins, the hitpoints and attack values of each of its units are reduced by ε. Then the simulation is run again
2. LOSS - if the team whose attribute values are to be perturbed losses, the hitpoints and attack values of each of its units are increased by ε. Then the simulation is run again
3. DRAW – in the case of draw the simulation stops. The new hitpoint and attack values are the outputs of the simulation. The unit mark-up can now be considered balanced.

Changing hitpoints and attack values is not the only option. A more sophisticated simulation would be able to change cooldown periods. Another improvement would be to change the values of individual units, instead of changing all units at once.

In addition, it is important to notice that this is not the only possible application of this method. Similar balancing techniques can be used to design units themselves. For example, a designer might want to know how many marines it takes to counter one tank.

The designer can provide the initial input by guessing the number of marines and the simulation can play out the game by increasing or decreasing the number of units until the game is a draw.

5. Experiments

5.1 Experiments Preview

Before describing in detail all the experiments that were performed to test our methods, we need to determine the kinds of experiments that are needed to test all proposed methods and their features. Our main goal is the creation of a comprehensive set of tests that will enable us to determine not only which method and parameter setting work the best, but also to gain insight into the advantages and disadvantages of each of the proposed methods and their suitability to different scenarios.

This, however, does not mean that we are going to tune any particular method (or make significant enhancements to it) to solve specific scenarios. We are going to leave the suggested methods as they are, giving them an equal opportunity to succeed, since our main goal is to understand the strengths and weaknesses of the proposed algorithms. The enhancements that are used, including move selection and better evaluation function, equally benefit most of the approaches, with the exception of the LP and Random players. Moreover, it will be difficult to test all possible situations, because there is an infinite number of ways to combining all the parameters; therefore, we will concentrate our efforts on solving several scenarios that are representative of the problems in the domain.

The selected scenarios are the 3 versus 3 and 4 versus 4 unit scenarios that belong to our original problem and also to the modified defensive version of the problem. In the defensive scenario the actions of players become more interdependent. This modification

is needed to test our hypothesis of whether RAB's performance is improved in situations where there is more interdependence between the actions of the two players.

In our first experiment we will examine the best way to assign scores for the RAB and the Monte Carlo methods. For each move these methods complete multiple runs, therefore the best move selection should be based on the combination of standard deviation and average scores for each such move. We test several such combinations to determine the scoring method used for the rest of the experiments. In the second experiment we compare the performance of the five proposed algorithms (i.e. RAB, Alpha-beta, Monte Carlo, LP, and Random) in typical combat scenarios. Our main goal is finding the best performer(s).

After such performers are determined the next step is to test the other search features and parameters that can be modified. Those include the evaluation functions (i.e. Simple, Square Root, and Tournament).

After such basic experiments are done we will have an idea of what method, evaluation function, and move selection setting are dominant and disregard the rest. At this point it will be possible to learn more about the advantages and the disadvantages of the best methods and to better understand in which cases one algorithm / setting is preferable to another.

First, an experiment to test the best method's accuracy needs to be performed. In such an experiment we will keep the number of nodes of the second best method constant, while increasing the number of nodes the best method visits. The performance of the best method should improve as the number of nodes given to it is increased. Another experiment is to test the performance of our algorithm under the strictest time constraints, giving an algorithm a very limited amount of nodes. This experiment is important since in a real-time domain we want algorithms to produce a good solution quickly.

Another experiment has to do with the move dependence in our domain. That is, we would like to see how the performance of RAB changes as the dependence between the players increases (i.e. going from the problem without defensive actions to the one with high degree of dependence). We would like to see whether in a defensive version of our problem, in which knowing / hypothesizing your opponents' moves can significantly improve the chances of winning, RAB will perform better as compared to the non-defensive problem.

Once the basic experiments are complete we will show the results for various improvements and problem extensions. One experiment will test whether restricting units to shoot at a specific target until that target is eliminated improves the performance as opposed to allowing units to re-aim at any time. Such a restriction can reduce the branching factor significantly; therefore, it is interesting to see how it affects performance, especially in situations where there are very few nodes available.

Next the problem extensions such as scenarios with range and the scenario balancing application will be tested. For the ranged scenario expansion we will test all of our methods to see whether their performance is similar to their performance on the original version of the problem. Finally, we will present a method for scenario balancing, which produces equal strength teams from originally unbalanced opponents by modifying the strength of one of the teams.

The next section describes the setup of the experimental environment, which is followed by a detailed description of each of the experiments that were performed.

5.2 Experimental Setup

A tournament style environment was set up for performing the experiments and for gathering statistics on the results. A tournament game is a match between two players who battle with each other until one player is eliminated. At each state both players

implement their respective algorithms to find their best move. Then both moves are executed simultaneously, the state of the game is updated, and the game continues until one or both players are eliminated (refer to Section 3 for a detailed example). Given two players *A* and *B*, a *win* for *A(B)* occurs when *A(B)* has unit(s) remaining while *B(A)* does not. A draw occurs when both players have no units remaining. Each experiment consists of *N* games. To make the summary of experiments easier to analyze and understand the experimental results were presented according to:

(#wins+0.5*#draws) / (#wins+ #losses+#draws)

The number of wins, losses, draws, as well as average scores achieved in each run is recorded. To minimize the variance symmetric starting positions are chosen.

The units in each team are generated randomly within predefined boundaries. There are three types of units: *tanks*, *marines*, and *artillery*. Each type has the ranges of *hitpoints*, *attack values* and *cooldown periods*. Shown in Table 2 below.

Table 2: Predefined ranges of hitpoints, attack values and cooldown periods for units used in our experiments.

	Tank	Marine	Artillery
Hitpoints	60 to 90	30 to 40	20 to 30
Attack Value	30 to 45	15 to 25	40 to 60
Cooldown period	1	0	2

Ranged units' locations are generated randomly within a square area. Then the *connectivity* of a given scenario was computed. If the connectivity is within 5% of the desired connectivity value (85% for our experiments), the tournament is run, otherwise

the random generation continued until the values are within the specified *connectivity value*.

To simulate real-time constraints, each player is given a limited number of resources. In order to ensure the comparability of experimental results that are run on different types of machines, the time given for each move is approximated by the number of nodes. The main reason for preferring node count over time is that machines with different CPU speeds and memory sizes perform different amounts of computations in a given time period. Approximating time as the number of nodes given per method ensures that similar amount of computation is performed. This holds true because there is no significant other overhead in all of the methods, therefore number of nodes counted is proportional to the execution time of the program.

For RAB (Random Alpha Beta) and AB (Alpha-Beta), which are search-based methods, the number of nodes consists of combination of interior nodes, cut nodes, terminal nodes, and leaf nodes. For the Monte-Carlo method the number of nodes consist of the total number of locations visited. The exceptions are the Random player and the LP player, which do not use number of nodes to approximate time. The Random player only visits one node every move, while the LP player has to perform a complete depth two search before it can produce a result. An LP player performing a complete depth two searches comes up with a solution within reasonable time frame, however increasing its search depth to four results in an explosion in the LP's execution time.

The actual node count limits that are used in the experiments were selected in order to produce acceptable real-time performance on the machines used for the experiments. Specifically, the experiments are run on Athlon MP / XP 2400+ to 2500+ processors with 512 MB – 1024 MBs of memory. For the experiments without defensive actions the node limit for one move in a game for each player when set to 200,000 nodes results in average game durations of ~4.5 seconds (Note: each non-defensive game consists of ~5-6 moves for each player). For experiments with defensive actions, when the node limit is set to

300,000 nodes, each game lasts on average ~9 seconds, consisting of ~6-7 moves for each player.

The two types of scenarios examined in most of the experiments are the 3 units versus 3 units and 4 units versus 4 units scenarios. The make up of the 3 vs 3 scenarios are two *marines* and one *tank*. In the 4 vs 4 scenarios there are one *artillery unit*, two *marines*, and one *tank*. In each game the unit parameters are generated randomly within specified ranges described in Table 2.

5.3 Results of Experiments

5.3.1 Experiment 1 (Scoring Setting for RAB and Monte Carlo)

In our first experiment we examine the best way to assign scores for the RAB and the Monte Carlo methods. For each move these methods complete multiple runs, therefore the best move selection should be based on the combination of standard deviation and average scores for each such move. We test several such combinations to determine the scoring method used for the rest of the experiments. The experiment is run for 3 vs 3 non-defensive scenario. Each experiment consists of 200 games. All players use the Square Root evaluation function and move selection with the limit of 15 moves. Both the RAB and Monte Carlo were tested against the AB player. The independent variable is the scoring setting, which consists of adding or subtracting a certain proportion of standard deviation from the average score. The results are presented in Figure 7.

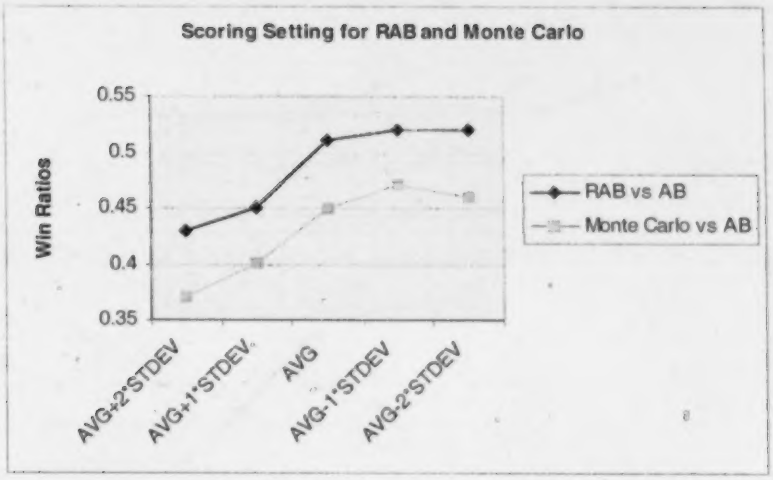


Figure 7: Win ratio of RAB over AB and Monte Carlo over AB for different scoring settings.

The results of this simple experiment show that the type of scoring setting used influences the performance of both RAB and Monte Carlo methods. Furthermore, they show that adding standard deviation to the average score results in significant decreases in performance for both RAB and Monte Carlo, compared to the method that uses only average scores. Subtracting standard deviation from the average results in a slight improvement in performance. For both the RAB and Monte Carlo the setting of (Average Score – 1 * Standard Deviation) produces the best results. Therefore, this scoring setting is used for these methods in the rest of the experiments.

5.3.2 Experiment 2 (Performance of All Methods)

The main goal of this experiment is to evaluate the relative performance of each method in two typical types of scenarios. The goal is to identify the methods that perform the best and then use those methods for further testing and discard the rest. The methods tested are: RAB, AB, Monte Carlo, LP and Random. We have run an experiment to test the performance of all the methods against each other on two scenarios, one without

defensive actions and the other one with a possibility to defend. Each experiment consists of 200 games, with randomly generated starting positions within specified boundaries. Each move is given 200,000 nodes for non-defensive scenario and 300,000 for defensive games. (Note: Defensive scenarios have higher branching factor and generally last longer, therefore more nodes are needed.) To make the experiment fair for all methods and also to simplify it, all methods use the same move selection values and the same evaluation function. Thus, all players (except Random and LP) use move selection with 15 move limit and Square Root evaluation function for the non-defensive scenarios. In the defensive scenarios the move selection setting is always 25. Each game is played between two equal teams of 3 units versus 3 units. Two of those units are marines and one is a tank. The results are presented in Figures 8 and 9 for non-defensive scenarios and in Figures 9 and 10 for defensive scenarios. Figures 8 and 10 show the cumulative win ratios achieved by each method over all other methods, while Figures 9 and 11 show the results of individual matches.

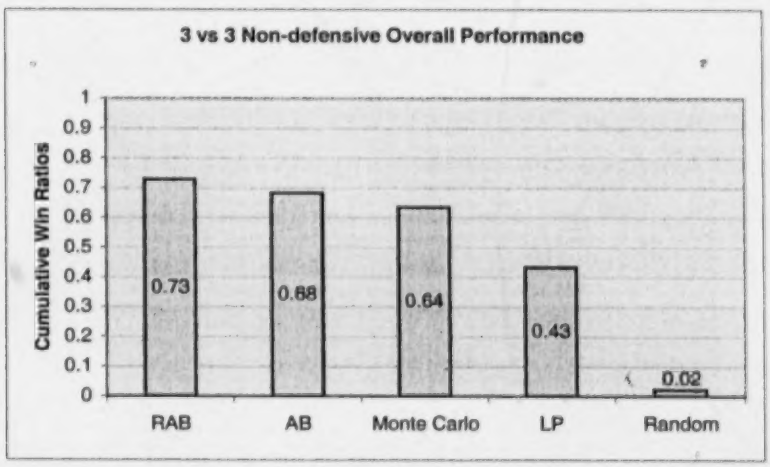


Figure 8: Win ratio for each individual method in a round robin tournament for a non-defensive scenario.

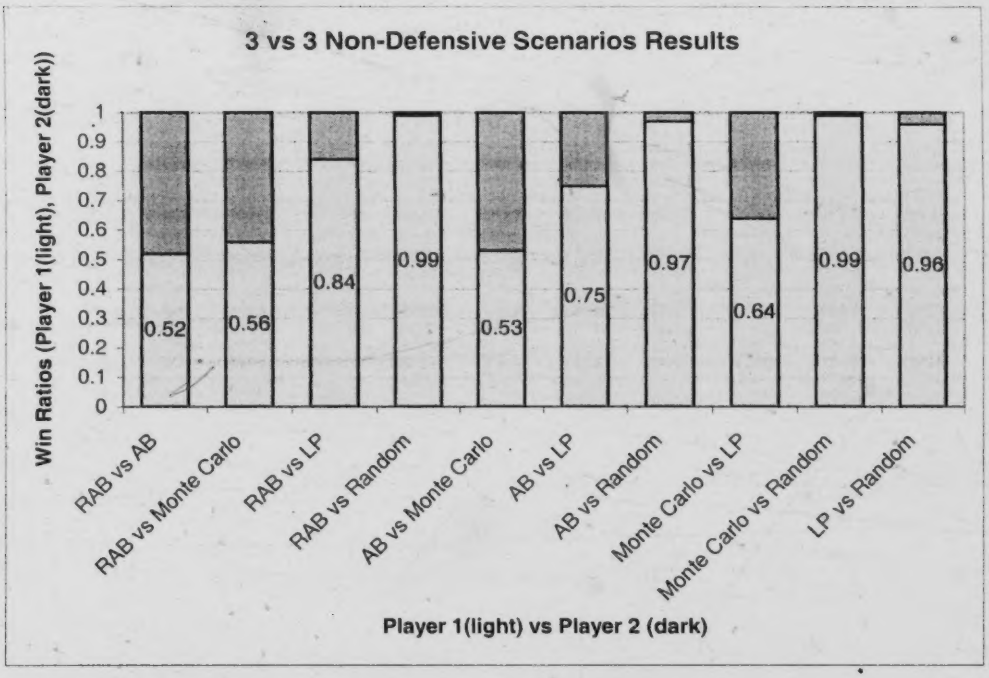


Figure 9: Results of all methods playing against one another in a non-defensive scenario.

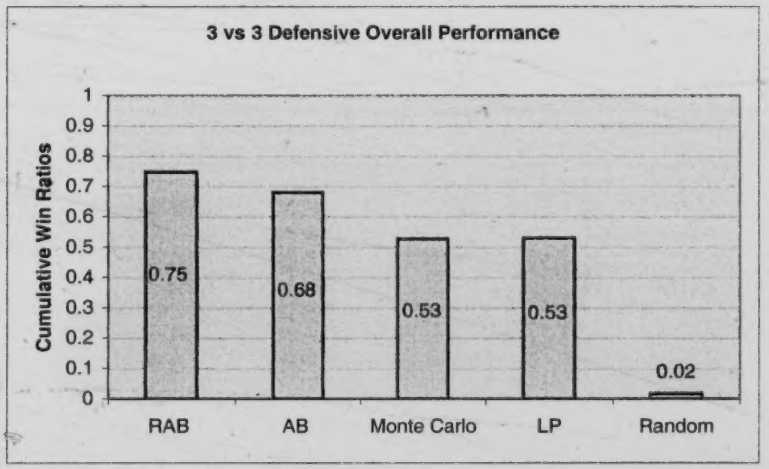


Figure 10: Win ratio for each individual method in a round robin tournament for a non-defensive scenario.

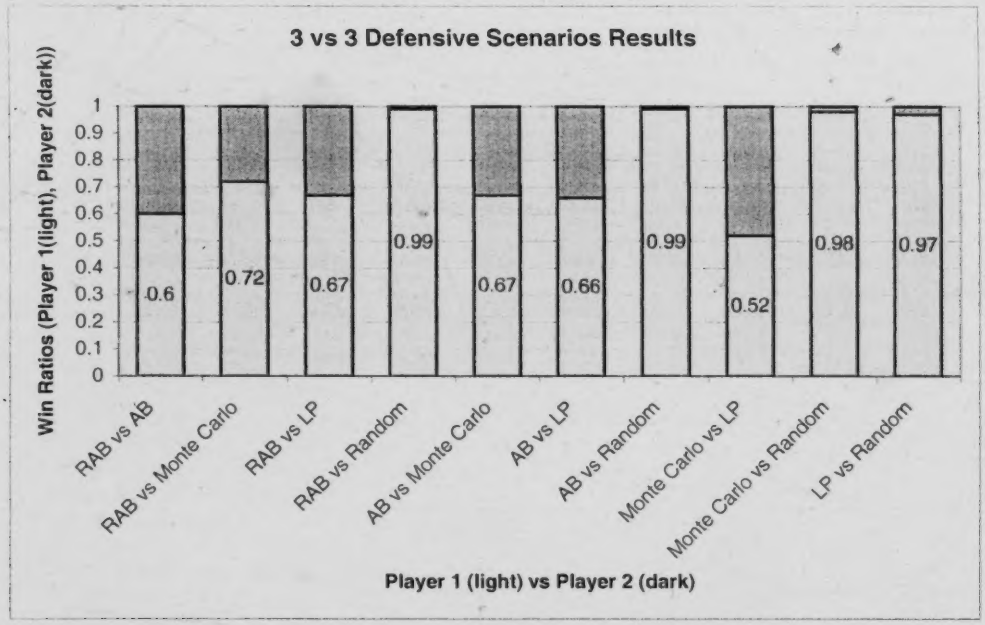


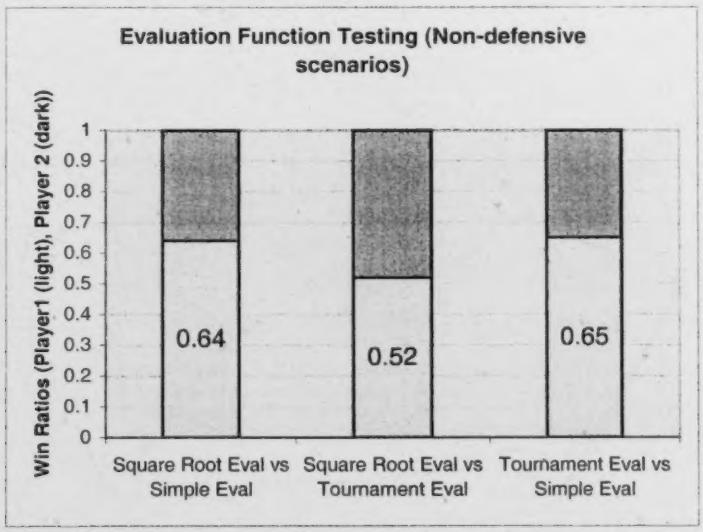
Figure 11: Results of all methods playing against one another in a defensive scenario.

The results of this experiment suggest that the RAB and Alpha-beta players are the best performers in two typical scenarios, with Monte Carlo coming third. The LP player's performance is not very close to that of the best methods, because of its limited search depth. But in more interdependent scenarios (Figure 10) the LP player's performance improves significantly, but still is not on par with that of either RAB or AB. Therefore, the rest of the experiments will concentrate on the RAB and AB methods to gain more understanding about their advantages and disadvantages.

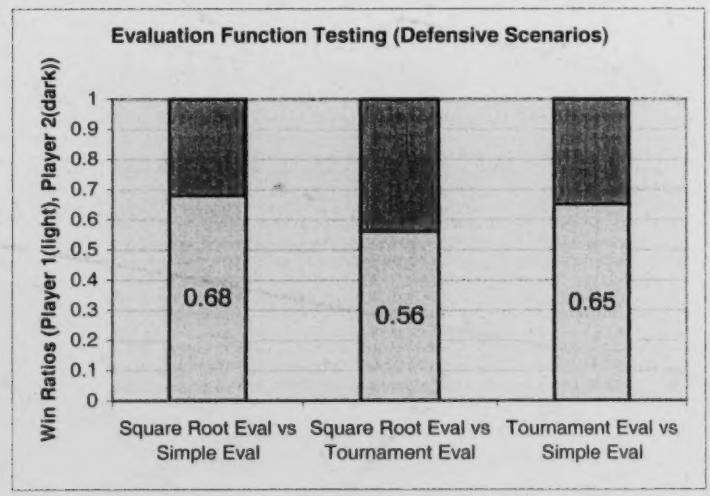
5.3.3 Experiment 3 (Evaluation Function Selection)

The goal of the third experiment is to determine the evaluation function that produces the best results. We test the following functions: Simple Evaluation, Square Root Evaluation, and Tournament Evaluation. For this experiment we only use the RAB method since its performance was determined to be the best. Each run consists of 200 games, with each

team given 300,000 nodes for scenarios with defensive actions and 200,000 nodes for scenarios without defensive actions. The scenario is the same as described in experiment 2 (i.e. two marines and one tank). The results are shown in Figure 12(a) for non-defensive scenarios and Figure 12(b) for defensive scenario.



(a)



(b)

Figure 12: Evaluation function performance: (a) non-defensive scenarios (b) defensive scenarios.

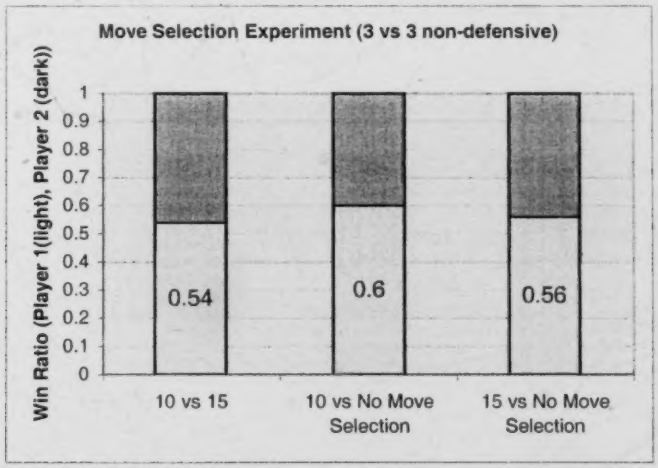
The results for both the defensive and non-defensive scenarios show that our initial hypothesis was incorrect and suggest that the Square Root evaluation function is better than the Tournament evaluation function and therefore it is the most suitable for our domain. This suggests that the extra computation involved in Tournament evaluation function does not pay; the Square Root evaluation function can complete more RAB iterations with the same number of nodes and it is a better choice. This is especially true for the defensive scenarios, where the Tournament evaluation function's games do not take into account the possibility of defensive actions. The Simple evaluation function was proven to be inferior by a significant margin to both the Tournament and the Square Root evaluation functions.

5.3.4 Experiment 4 (Move Selection Experiment)

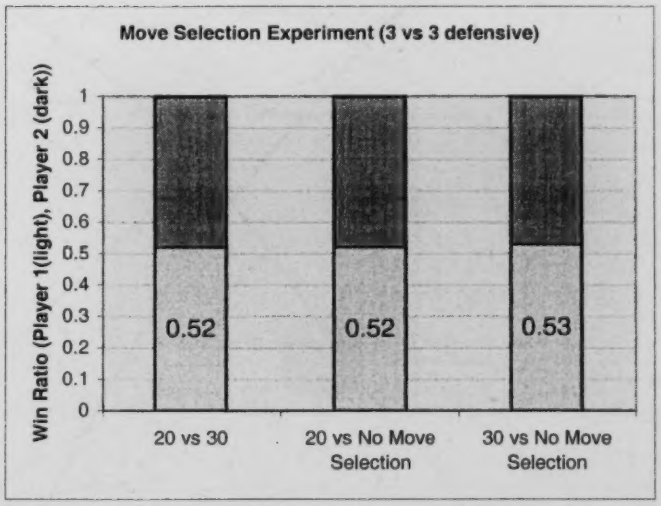
Another key feature that needs to be tested is the number of moves to be considered at each level. As a reminder, in our methods, move selection is executed by performing depth one greedy executions of all moves and selecting the top N moves based on the results returned by the evaluation function. Therefore, it is important to determine which evaluation function performs the best (i.e. has the best future prediction), before we can find the optimal number of moves to be selected for a given scenario. Also, the number of moves to be selected is likely dependent on the type of scenario at hand and is expected to be proportional to the branching factor of a given scenario. Because the number of moves in 4 vs 4 scenarios is exponentially larger than that in the 3 vs 3 scenarios, it is likely that there are many more good moves that are close to each other. Thus, in a 3 versus 3 scenario the optimal number of moves to be selected is expected to be smaller than in 4 versus 4 scenarios.

For this experiment we are using only the RAB method applied to defensive and non-defensive scenarios. There are two problem scenarios: 3 versus 3 and 4 versus 4. For non-defensive scenarios each team gets 200,000 nodes for 3 vs 3 scenarios and 400,000 nodes

for 4 vs 4 scenarios. For defensive scenarios the number of nodes is 300,000 and 600,000 respectively. The evaluation function used is the one determined to be the best in Section 5.3.3 (i.e. Square Root Evaluation). Each run consists of 200 games. The results are shown in Table 13 for 3 vs 3 scenarios and Table 14 for the 4 vs 4 scenarios.

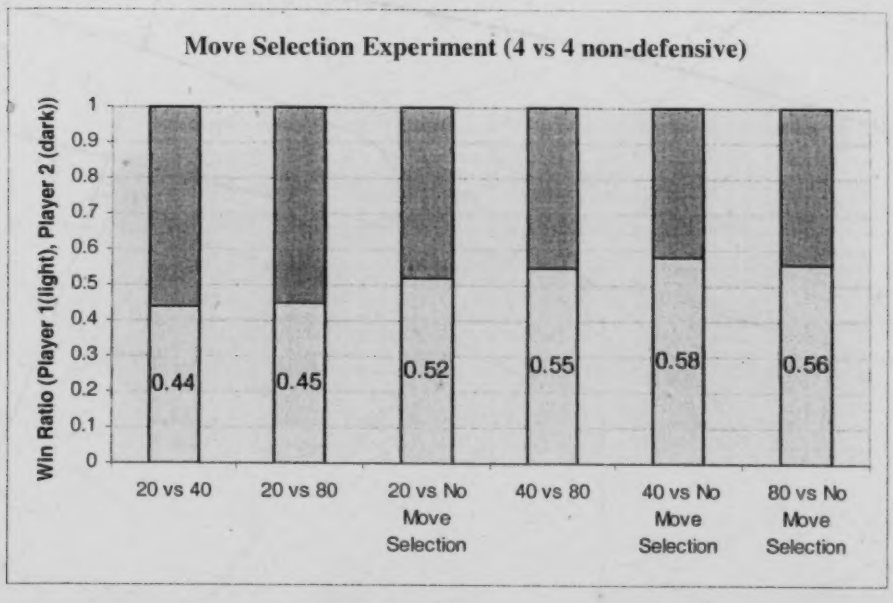


(a)

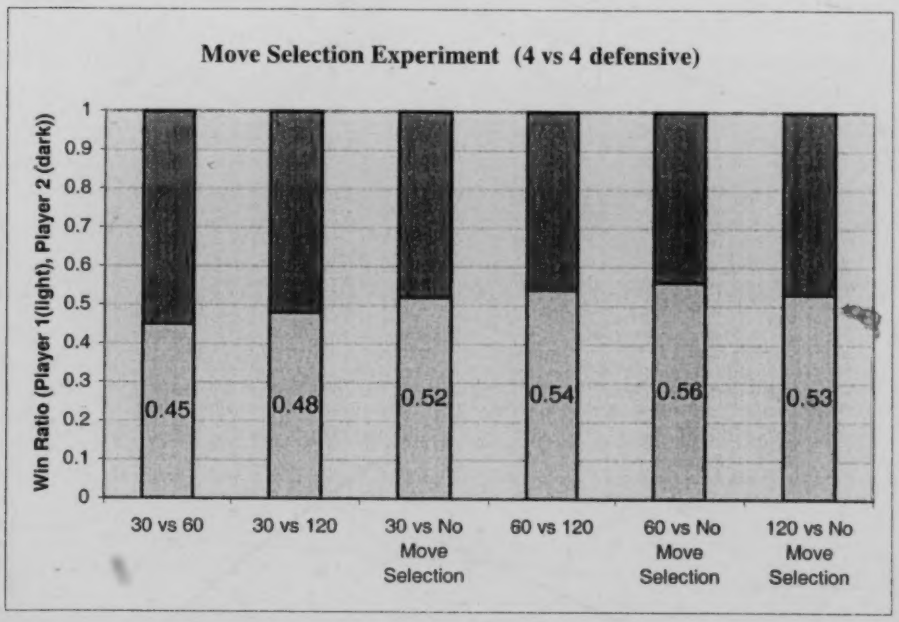


(b)

Figure 13: Move selection results in 3 versus 3 non-defensive (a) and defensive (b) scenarios. Results show that concentrating effort on a subset of moves payoffs.



(a)



(b)

Figure 14: Move selection results in 4 vs 4 non-defensive (a) and defensive (b) scenarios. Results show that concentrating effort on a subset of moves payoffs.

In general, the results for the move selection experiments show that in all situations some degree of move selection is preferable to having no move selection. This means that it pays off to concentrate search effort on a small subset of moves that can be searched deeper, instead of searching all moves to a lower depth. For the Random Alpha-Beta method it means that it is beneficial to invest more nodes into more iterations for more accurate predictions rather than into exploring more moves. Also, it shows that our Square Root evaluation function is successful in selecting good moves based only on depth one look-ahead search. Thus, for 3 vs 3 non-defensive and defensive scenarios the move selection is 10 and 20, respectively, while for 4 vs 4 non-defensive and defensive scenarios it is set to 40 and 60, respectively.

5.3.5 Experiment 5 (Varying RAB's Nodes vs Constant AB)

This experiment is designed to determine whether the quality of RAB's results improves as the algorithm is given more nodes. In this experiment RAB is playing against the AB algorithm whose maximal node count is held constant and the number of nodes assigned to RAB is varied. The experiment is performed on 4 representative scenarios: 3 vs 3 non-defensive, 3 vs 3 defensive, 4 vs 4 non-defensive, and 4 vs 4 defensive. For 3 vs 3 defensive and non-defensive scenarios AB is assigned 50,000 and 100,000 nodes, respectively. For both 4 vs 4 defensive and non-defensive scenarios AB is assigned 200,000. The evaluation function and move selection values used are the best determined in experiments three and four, respectively.

The results are shown in Figures 15 and 16 for non-defensive and defensive scenarios, respectively. The general trend is that as the number of nodes increases the quality of RAB solutions increases. The increase is more gradual in the case of defensive scenarios, as compared to the non-defensive ones. In the case of non-defensive scenarios (specifically the 4 vs 4 non-defensive scenario) there is a big improvement in performance from 100,000 to 200,000. From examining the solutions it seems that such improvement is due mostly to the higher depth RAB reaches when given 200,000 nodes.

In non-defensive scenarios there is less interdependence than in the defensive scenarios, therefore reaching higher depths has more effect on the quality of the resultant solution. This is further supported by the gradual increase when moving from 200,000 to 600,000, when the improvement reached its peak.

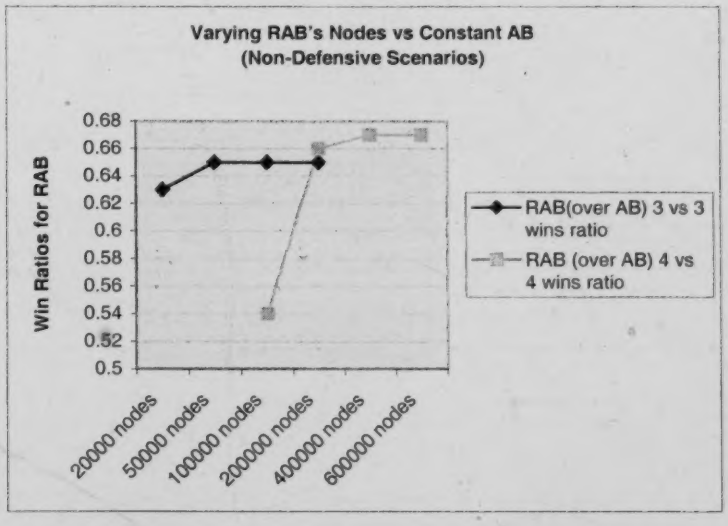


Figure 15: Results of RAB wins over AB as the number of RAB nodes is increased from 20,000 to 400,000 and AB's node count is fixed. The results are shown for non-defensive scenarios.

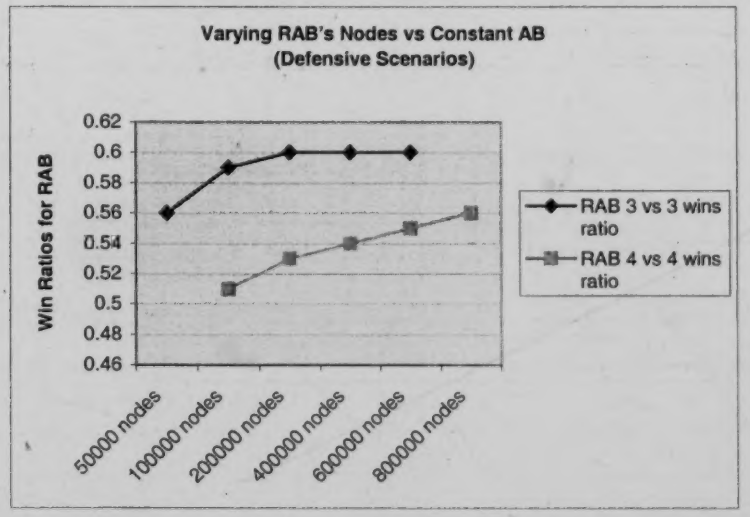


Figure 16: Results of RAB wins over AB as the number of RAB nodes is increased from 50,000 to 400,000 and AB's node count is fixed. The results are shown for defensive scenarios.

5.3.6 Experiment 6 (Strict Constraints)

The abstract combat domain is most and foremost a real-time domain, because in actual RTS games AI is allotted only a small percentage of CPU cycles. Therefore, it is very important to see how the algorithms perform under the strictest time constraints and whether their performance changes as more resources become available to them. For this experiment we are using the RAB and AB methods with the best setting from previous experiments (i.e. best evaluation function and the best number of moves). The experiment is run for 3 vs 3 and 4 vs 4 non-defensive and defensive scenarios. Each run consists of 200 games. The independent variable is the number of nodes given to each method. The Square Root evaluation function is used as well. The results are shown in Figure 17 for non-defensive scenarios and Figure 18 for defensive scenarios.

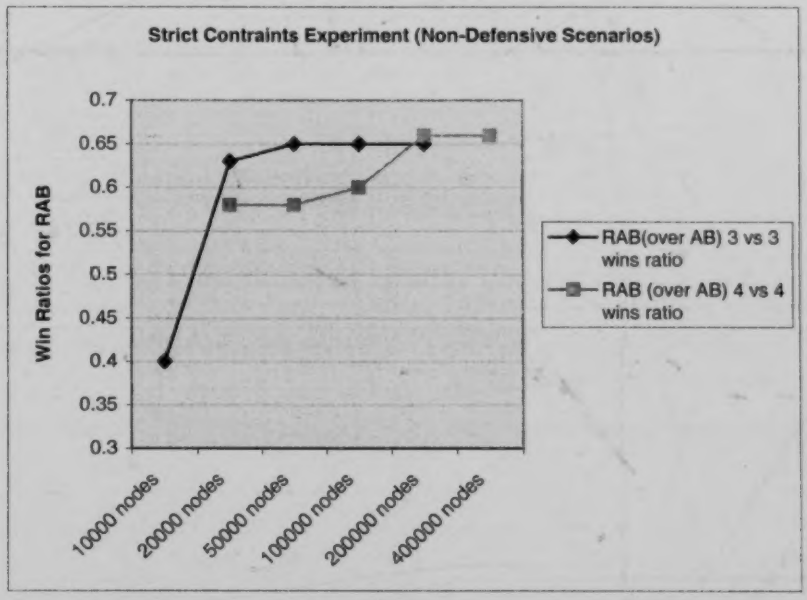


Figure 17: Results of RAB wins over AB as the number of nodes is increased from 10,000 to 100,000 for 3vs3 scenarios and from 20,000 to 200,000 for 4vs4 for non-defensive scenarios.

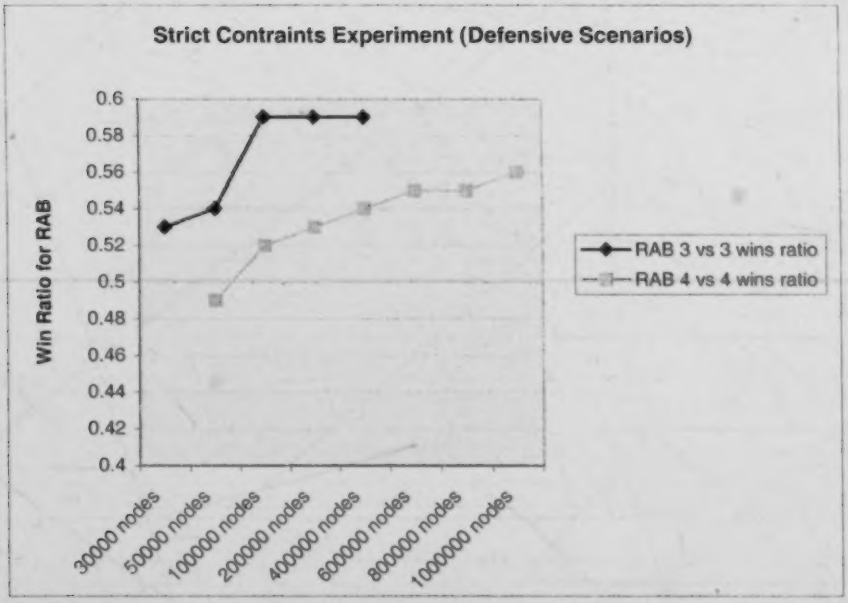


Figure 18: Results of RAB wins over AB as the number of nodes is increased from 30,000 to 200,000 for 3vs3 scenarios and from 50,000 to 400,000 for 4vs4 for defensive scenarios.

The results for both defensive and non-defensive scenarios show that RAB performs better than AB across most of the settings. The most surprising finding is that given a very limited number of nodes for both defensive and non-defensive scenarios RAB outperforms AB. This shows that even though AB can reach greater depth than RAB with the same node limit, investing into randomization and extra runs rather than into deeper searches pays off very early for the RAB algorithm. Another general trend observed is the gradual reduction of RAB's improvement over AB. The results show that as the number of nodes increased for both algorithms RAB reaches a ceiling in its winning percentage over AB.

5.3.7 Experiment 7 (Degree of Move Interdependence)

Our initial hypothesis was that the original abstract combat scenarios (i.e. no defensive action) do not feature a high interdependence between opponent's moves. That is, it is not very important for a player to know what the opponent will do. With the introduction of the defensive action the outcome can become highly dependent on the player's ability to guess correctly (or estimate) what the opponent's action will be. Therefore, our expectation is that RAB will improve its performance as the degree of dependence increases. We think that in highly interdependent domains the search depth, which is RAB's main weakness, is not as important as its ability to estimate what the opponent will do. In such situations estimating correctly which move the opponent is likely to choose immediately, can give a player a significant advantage.

The RAB and AB methods will be used in this experiment. The independent variable will be the degree of dependence of a given scenario. This can be easily adjusted in our domain starting with a setting with no defensive actions, and finishing with the setting where the probability of selecting a defensive action is similar to that of selecting an offensive action. Specifically, in a scenario with no reward for the defensive actions units are not motivated to execute the defensive actions since the defensive actions do not benefit them. Then the reward for choosing a defensive action increases, by increasing the weight of ratios of attack values assigned for defence and for hitting back at the attacker, until the likelihood of selecting a defensive action is similar to that of selecting an offensive action. For 3 vs 3 scenarios 300,000 nodes are given for one search. For 4 vs 4 500,000 nodes are provided. The number of moves selected are 20 and 50 for 3 vs 3 and 4 vs 4, respectively. The Square Root evaluation function is used. The results are shown in Figures 19 and 20 for 3 vs 3 and 4 vs 4 scenarios, respectively.

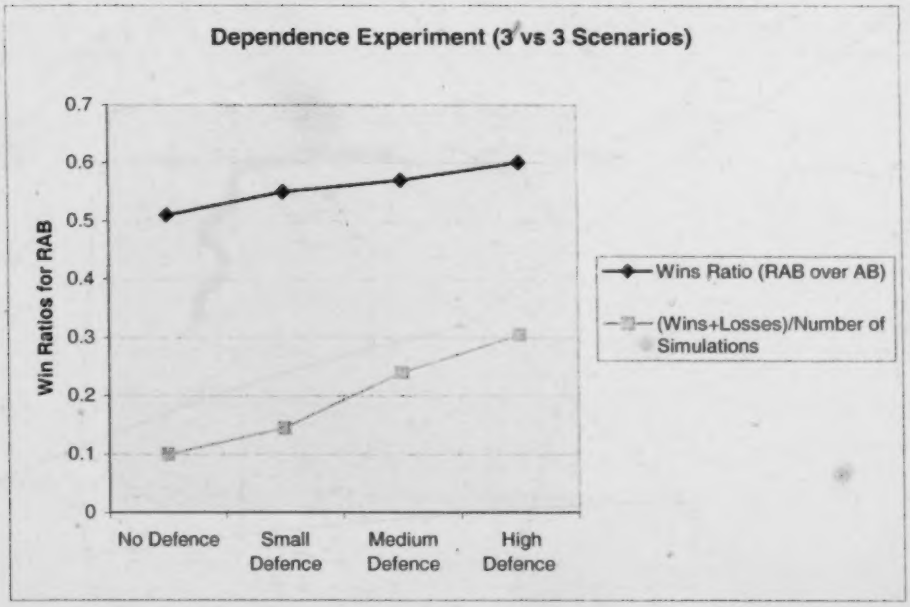


Figure 19: The ratio of wins of RAB over the AB for 3 vs 3 scenarios as the move dependency increases from no defensive action to high probability of a defensive action. The figure also shows the number of wins/losses as a percentage of games played in such situation, indicating result variance.

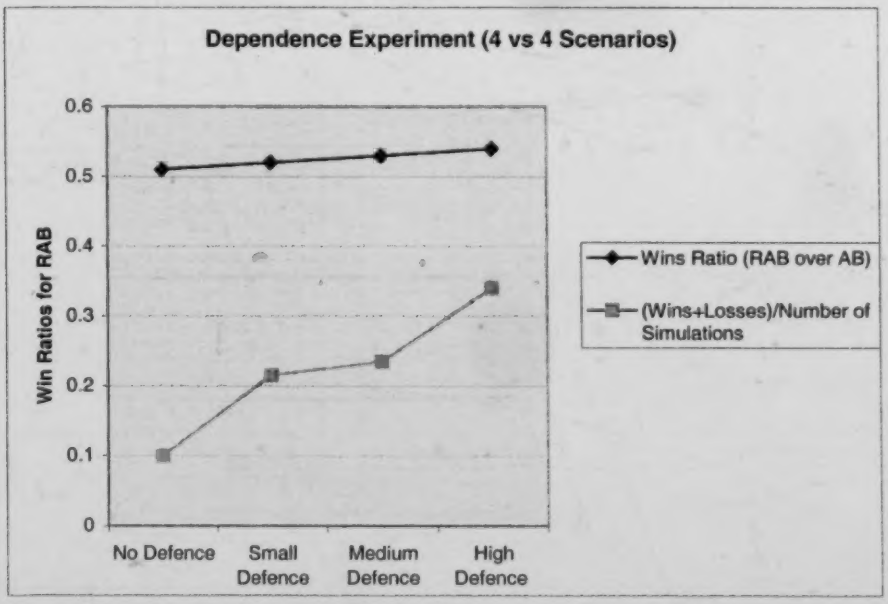


Figure 20: The ratio of wins of RAB over the AB for 4 vs 4 scenarios as the move dependency increases from no defensive action to high probability of a defensive action. The figure also shows the number of wins/losses as a percentage of games played in such situation, indicating result variance.

The results for both 3 vs 3 and 4 vs 4 situations show that as the tendency to pick a defensive action increases the win ratios for RAB over AB increase in both the 3 vs 3 and 4 vs 4 case. This result underpins our initial hypothesis that in highly interdependent scenarios the RAB will perform better. Another correlation that can be observed in both graphs is between the win ratio of RAB and the number of wins and losses as a percentage of the number of simulations. This is not surprising, since as the move interdependence increases the success of actions increasingly depends on what the opponent will choose to do. Therefore, in a highly defensive scenario there is no single move that guarantees at least a draw for a player. The opponent can counteract most moves taken by the player leading to a higher standard deviation of the results.

5.3.8 Experiment 8 (Sticking to Target Improvement)

One of the constraints that can significantly reduce the branching factor is not allowing re-aiming. It means that if a unit has picked a target it should keep shooting (stick) at that target. That is, from the time the unit has picked a target until the target elimination, that unit has only one action available to it. We would like to see whether not allowing units to re-aim could lead to a better real-time performance.

In this experiment we use our best method (RAB), in combination with re-aiming in the first case and without re-aiming in the second case. The experiments are performed for 3 vs 3 and 4 vs 4 non-defensive scenarios. For all cases the Square Root evaluation function is used. The move selection number is 10 for 3 vs 3 non-defensive scenarios and 40 for 4 vs 4 non-defensive scenarios. The independent variable is the number of nodes given for each search. Each run consists of 200 games. The results are shown in Figure 21.

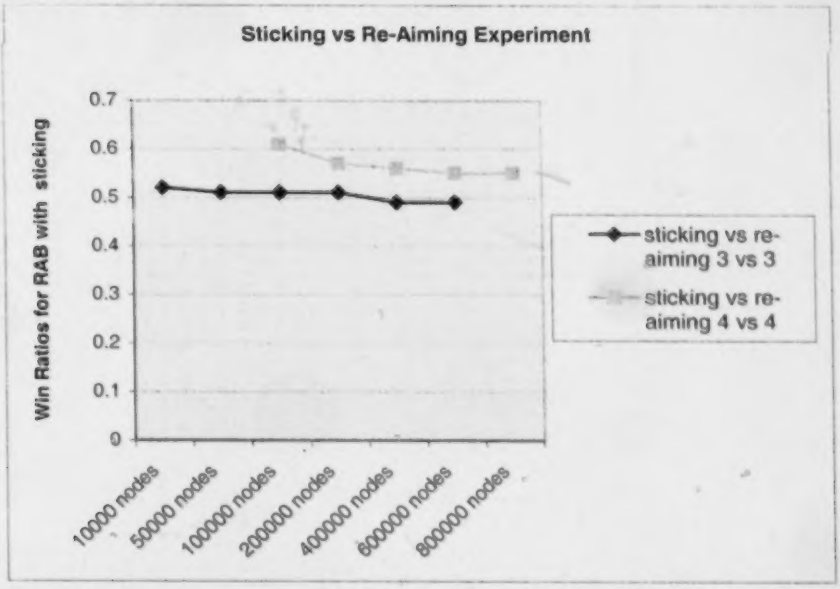


Figure 21: The percentage of wins for RAB in 3 vs 3 and 4 vs non-defensive scenarios as the number of nodes given is increased from 10,000 to 800,000.

The results show a small advantage when units implement the sticking to the target policy. Because of the reduced branching factor when no re-aiming is allowed it is more advantageous to use no re-aiming when the number of nodes given is small. As the number of nodes given increases the performance of the method that does not stick to its target slowly increases to over 0.5 in the 3 vs 3 case. We can conclude that not allowing re-aiming is especially useful when there are strict real-time constraints, however when the node limit is increased the performance of the no-sticking algorithm improves.

5.3.9 Experiment 9 (Examining the Range Expansion)

This experiment is similar to the second experiment except that it tests the performance of various methods on scenarios where range is included. The experiment is performed only on the non-defensive scenarios because during ranged scenarios not all units are within reach of all other opponent units. This situation can increase the likelihood of deadlocks occurring, where units that are within reach of each other use only defensive

actions. In our application we do not allow all units to take a defensive action at the same time to avoid deadlocks. Therefore, in a situation where a unit is the only one that can attack a target or a group of targets it will not be allowed to perform a defensive action. Thus, in a scenario where range is considered due to such restrictions only a few defensive actions will be selected, which will make the move selection there comparable to that of non-defensive scenarios.

The methods tested are RAB, AB, Monte Carlo, LP and random. We run a round robin tournament on two scenarios: 3 vs 3 units and 4 vs 4 units. Each experiment consists of 200 games, with randomly generated starting positions within specified boundaries. The boundaries are defined by a square of size 20 by 20. A game only takes place when its randomly generated *connectivity value* (refer to Section 3.3) is within 0.05 of 0.85. Each move is given 200,000 nodes for 3 vs 3 scenarios and 400,000 nodes for 4 vs 4 scenarios. To make the experiment fair for all methods and also to simplify it, all methods use the same move sorting values and the same evaluation function. Thus, all players (except Random and LP) use move selection of 20 and 40 moves for 3 vs 3 and 4 vs 4, respectively and the Square Root evaluation function. In 3 vs 3 scenarios there are two marine units and one tank unit. For 4 vs 4 scenarios an artillery unit is added to the setup. The results are presented in Figures 22 and 23 for 3 vs 3 scenarios and in Figures 23 and 24 for 4 vs 4 scenarios. Figures 22 and 24 show the cumulative win ratios achieved by each method over all other methods, while Figures 23 and 25 show the results of individual games.

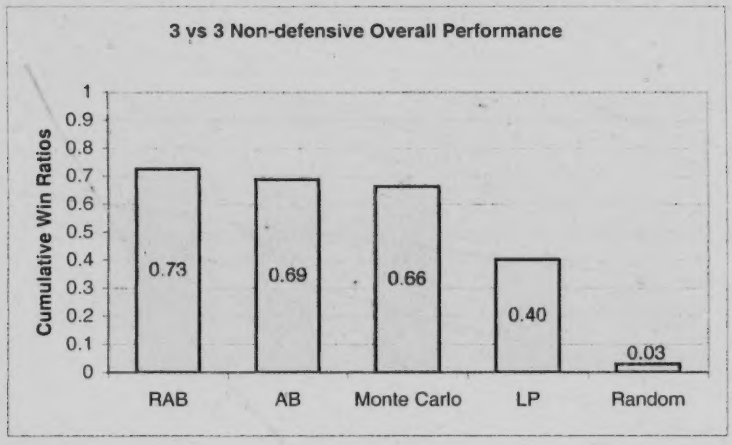


Figure 22: The combined totals of win ratio achieved over all methods for each individual method in non-defensive scenarios with range.

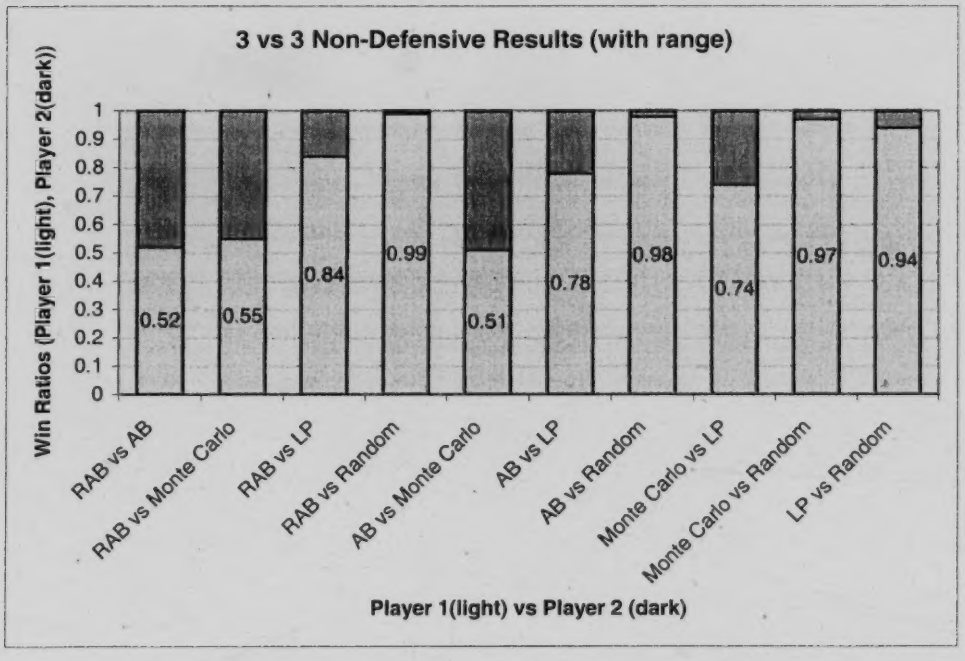


Figure 23: Results for ranged non-defensive scenarios in 3 vs 3 situations.

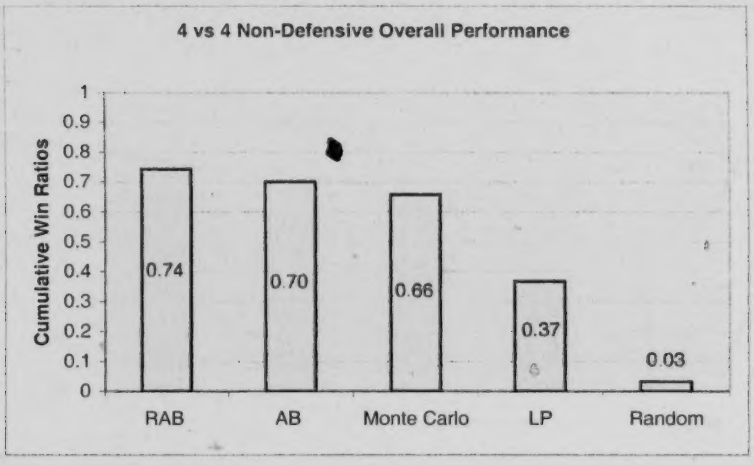


Figure 24: The combined totals of win ratio achieved over all methods for each individual method in non-defensive scenarios with range.

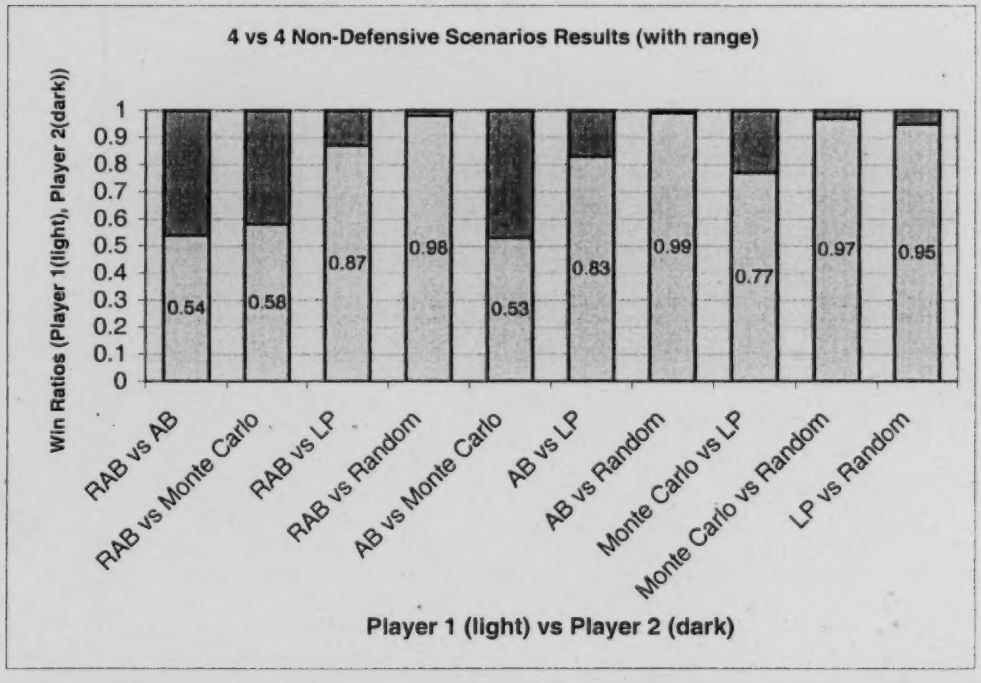


Figure 25: Results for ranged non-defensive scenarios in 4 vs 4 situations.

The results for experiments in both the 3 vs 3 and 4 vs 4 situations show that again RAB has a small edge over AB. The performance of Monte-Carlo and LP players is similar to that in the experiments where range was not considered. So our hope that the RAB performs better than other methods in scenarios with range expansion was confirmed in this experiment.

5.4 Unit Value Balancing

To test the simple version of the balancing application we have run two sample games: one for the defensive scenarios, while the other for the non-defensive scenarios. For each game we set up two teams. The first team consists of units that cannot be changed while the second teams' unit strengths can be increased or decreased depending on the results of the previous game. For both the defensive and non-defensive scenarios the constant team consists of one tank and one marine. The second team to be changed consists of three marines. Thus, our primary goal is to change the values of the second team's units in such a way that they are equal opponents to the first team. The initial set-up for the two teams is as follows:

	Unit 1	Unit 2	Unit 3
Team 1	H: 60 A: 30 C: 0/1	H: 25 A: 45 C: 0/2	
Team 2	H: 35 A: 20 C: 0/0	H: 35 A: 20 C: 0/0	H: 35 A: 20 C: 0/0

The results after balancing was performed are shown in Table 3.

Table 3: Shows the results for team 2 after balancing for non-defensive (a) and defensive scenarios (b)

	Unit 1	Unit 2	Unit 3
Team 1	H: 60 A: 30 C: 0/1	H: 25 A: 45 C: 0/2	
Team 2	H: 30 A: 15 C: 0/0	H: 30 A: 15 C: 0/0	H: 30 A: 15 C: 0/0

(a)

	Unit 1	Unit 2	Unit 3
Team 1	H: 60 A: 30 C: 0/1	H: 25 A: 45 C: 0/2	
Team 2	H: 29 A: 14 C: 0/0	H: 29 A: 14 C: 0/0	H: 29 A: 14 C: 0/0

(b)

The results show that the hitpoints / attack values for the team of marines was reduced to (H:30 A: 15) for each marine in non-defensive situation and (H:29, A:14) for the defensive scenarios. The results for both the defensive and non-defensive scenarios are very close. The slight difference between the two scenarios suggests that some units, such as marines, that have a lower ratio of attack value over hitpoints are better suited for scenarios with defensive actions. Units such as artillery and tanks that have a higher ratio are more reluctant to defend due to the risk of wasting their higher attack value.

6. Conclusion

One of the goals of our research was to examine whether search-based methods can be used effectively in real-time domains with simultaneous move execution. In our research we have overcome many challenges. We have shown that search-based methods can be successful in a domain with the real-time constraints and with move interdependence. Moreover, we have shown that non-deterministic search methods such as RAB perform better in such situations than a traditional alpha-beta algorithm. This result shows that the search depth is not the most essential feature when designing algorithms in simultaneous

move domains, as opposed to alternating move complete information domains, where search depth was shown to strongly correlate with the quality of the resultant solution.

In the process we have also varied the settings of problems in our simultaneous move abstract combat domain and have shown that there is no single best method for all simultaneous move scenarios. The key to solving simultaneous move scenarios effectively is through examining the degree of interdependence between the opponents' actions. Some domains that have simultaneous move execution have very low degree of move interdependence and therefore in those domains the deterministic methods can perform well. In cases with higher degree of interdependence our RAB method was proven to be the most suitable real-time algorithm.

We have also achieved more practical goals. We have created an experimental environment where methods can be tested and improved, algorithms can be added and parameters can be adjusted with minimal effort, by playing tournament games against other methods. Moreover, the object-oriented structure of our software ensures that it can be expanded to work in other simultaneous and alternating move domains with minimal adaptations required.

Our main goal, however, was to create effective real-time algorithms for abstract combat scenarios. We have shown that three of our proposed methods (i.e. AB, RAB, Monte Carlo) can provide reasonable anytime solutions for various scenarios in the abstract combat domain. Out of these the RAB algorithm was proven to be the most solid performer since in most situations it produced the best overall results.

7. Future Work

The goal of our work was to examine how abstract combat problems can be solved effectively in real-time. We have shown that search-based approaches can be used

effectively. In the process of tackling our problems, we have discovered numerous problem areas that were not intended to be researched initially and have raised many challenging questions that could not be addressed in this research.

One such issue is that of determining or estimating the degree of move interdependence in the games with simultaneous move execution. Specifically, we think that in non-defensive scenarios there is very little dependence between the players' actions meaning that alpha-beta can be used there almost as effectively as RAB, while the opposite is true for the defensive scenarios. However, we have no theoretical proof for that. Such a proof together with a tool that can estimate a degree of dependence in a given domain could be very useful in helping determine the kinds of algorithms suitable for different simultaneous move domains. For example, a simultaneous move scenario that has no dependence between opponent's moves will be solved most effectively by the alpha-beta algorithm without the need for randomization.

Our RAB algorithm can also be a focus of future work. The current version has one problem, which arises because the successors of the root node in the RAB tree are not randomized, that is there RAB behaves just like AB. For instance, in a game of Rock-Paper-Scissors, which is at a maximum a depth=2 game, the player executing the search will always lose, since the opponent at all times can select a move that counteracts that player's move. The same applies to our defensive scenario, where most pure strategies can be defeated by other pure strategies. In such a situation it is often sensible to model what move the opponent is likely to choose and then select a move that counteracts the opponent's move. Such a strategy might be riskier but it can also pay off. Since there is no guarantee that one move will lead to a win, it might be useful to select several promising moves and pick one of them each turn. This weakness of RAB can be addressed by performing two searches and then combining their results. The first search is performed where the player to move starts at the root, while the second search is performed where the opponent starts at the root. The top moves for the player to move from each of the searches are then selected and one of these move is executed according to certain probability distribution.

Another area of interest is improving algorithms to solve more complicated problems in the abstract combat domain. The best proposed algorithm (i.e. the RAB) is currently capable of providing quality solutions only in smaller-scale combat scenarios. But as the number of units in a scenario grows the branching factor increases exponentially. Therefore, given similar resources the quality of solutions produced by our method will decrease. This impact is somewhat softened by the fact that the evaluation function is capable of successfully selecting only a fraction of generated moves for future examination. Also, as the number of units increases the branching factor does not seem to be exponential (i.e. for 3vs3 the best move selection setting is 10 moves, while for 4vs4 the best setting is only 40 moves). Nevertheless, the solution quality still decreases as more units are added, requiring either more resources, which can make algorithms to become non-real-time, or acceptance of poorer quality solutions with unchanged amount of resources.

There are at least two ways of tackling the abovementioned problem. First, we can aggregate a number of units into a single or multiple *super-units*. So, given any number of units, the number of aggregated units is always reduced to a size that the algorithm can manage in real-time. Such approach does guarantee an upper bound on the time required to produce a solution for a scenario of any size, but it also has some significant drawbacks. For one, lumping many units into one bigger unit restricts all the units belonging to a single *super-unit* to attack only a single opponent *super-unit*. Another major problem is that of distributing the firepower between the opponent's units that belong to the *super-unit*. The questions that arise here are whether every unit should attack different opponent unit or whether they should all concentrate fire on one or several of the opponent's units belonging to that *super-unit*. We have tried implementing a simple aggregation approach with mixed success, which did not adequately address those problems. In the process, we have realized that successfully implementing an effective aggregation application would require algorithms that go beyond the scope of the intended research in this thesis.

Another way of addressing the multi unit problem is to divide units into independent groups with hierarchical top-down control. In other words, instead of simulating a single large battle scenario, we propose simulating a series of smaller localized battles. Each group involved in a small battle is only concerned with that particular battle, which ensures that search is performed within given time frame. Up the hierarchy is the commander whose responsibility is to decide on how to create such (player, opponent) groupings. Making such grouping decisions to effectively simulate the real situation is the key challenge. In scenarios, where weapon range and movement are considered the grouping decisions can be based on proximity of a unit to its opponents and its range.

One more interesting area for future research is that of extending our ranged scenarios to include unit movement. Such an extension would make the abstract combat domain closely resemble the realistic environment of both real-time strategy games and real military combat. Such an extension is, however, the most difficult challenge to tackle, since adding the possibility of movement increases the branching factor tremendously. But instead of performing a complete search in all possible move situations we propose performing complete searches only on selective possible future states. The player performing the search can select a certain number of promising moves using a good move selection technique. Then for each such move a number of promising opponent moves can be considered. After that a search can be performed for each of these situations and statistics for each of the players' moves can be gathered (Note: this is very similar to how current random alpha-beta works) and the best move will be selected. The key challenge here is coming up with a successful move selection technique that can reduce the large number of possible states to a few likely candidates. Such a technique will be more difficult to design than the move selection for the scenarios we studied, because of the two added degrees of freedom (i.e. the range and movement parameters).

The last area where research can be concentrated has to do with balancing units. This application is more of an engineering problem than a research issue but it is still an important area because game designers can use this application almost immediately. Currently, the balancing is done in a primitive way by adjusting the hitpoints and attack

values of all units in a team by ε until the balance is achieved. The application can be made more comprehensive by being able to adjust the attributes of a single unit and by also adjusting the cooldown periods. Another possibility is extending the application to game length balancing. This can be useful when trying to fix the number of turns, which approximates the time-length it takes one team to eliminate another team. In a typical situation, a game designer would like to create teams in a way that the human player's units will win over the computer player's units, but the fight has to last at least N turns.

Our next step will be to incorporate our methods into an ORTS [8] client either as a part of architecture of a standalone AI player or as a helper for a human player. There they can be used to alleviate the human players from the command of all the details and as an actual AI technique for resolving smaller scale battles for computer players.

8. References

[1] P. Allen. Situational force scoring: Accounting for combined arms effects in aggregate combat models, N-3423-NA, *RAND*, 1992

[2] H. Berliner. Backgammon computer program beats world champion. *Artificial Intelligence*, Vol. 14 pp. 205-220, 1980

[13] B. Bouzy. Associating domain-dependent knowledge and Monte Carlo approaches within a go program, *Joint Conference on Information Sciences*, Cary 2003, pp 505-508, 2003

[4] D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Approximating game-theoretic optimal strategies for full-scale poker, *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 661-668, 2003

[5] J. Bracken, J. Kress and R. Rosenthal. Warfare Modeling, published for the Military Operations Research Society, John Wiley and Sons, Danvers, Mass, 1995

[6] A. Brudno. Bounds and valuations for abridging the search for estimates. Problems of Cybernetics. Vol. 10 pp. 225-241, 1963. Translation of original Russian version from Problemy Kibernetiki.

[7] M. Buro, Improving Heuristic Mini-Max Search by Supervised Learning , *Artificial Intelligence*, Vol. 134 (1-2) pp. 85-99, 2002

[8] M. Buro, ORTS: A hack-free RTS game environment, *International Computers and Games Conference*, Edmonton, Canada, pp. 156-161, 2002

[9] M. Buro. Solving the Oshi-Zumo game, *Advances in Computer Games Conference 10*, Graz, pp. 361-366, 2003

[10] M. Buro. Real time strategy games: A new AI research challenge, *International Joint Conference on AI*, Acapulco, Mexico, pp. 1534-1535, 2003

[11] M. Buro. NSERC Reseach Proposal, Form 101, 2002

[12] B. Caldwell, J. Hartman, S. Parry, A. Washburn, and M. Youngren. Aggregated combat models, *Operations Research Department, Naval Postgraduate School Notes*, 2000

[13] S. Clemens, The application of Lanchester models to the battle of Kursk, Unpublished manuscript, Yale University, 1997

[14] T. R. Dupuy. Attrition: Forecasting battle casualties and equipment losses in modern war. *Nova Publications*, 1995

[15] R. Gozel. Firepower score attrition algorithms in highly aggregated combat models. *RAND*, pp. 47-60, 2000

[16] J. Haugeland. Editor. Artificial Intelligence: The very idea. MIT Press, Cambridge, Massachusetts, 1985

[17] T. Hauk, M. Buro, and J. Schaeffer. Minimax performance in backgammon. *Computers and Games Conference*, Israel, pp. 178-185, 2004

[18] F. Hsu. Behind Deep Blue: Building the computer that defeated the world chess champion. Princeton University Press, 2004

[19] W. P. Hughes, Editor. Military modeling for decision making. MORS Publications, third edition, 1997

[20] R. E. Korf. Real-time heuristic search, *Artificial Intelligence*, Vol. 42 (2-3) pp. 189-211, 1990

[21] D. Knuth and R. Moore. An analysis of alphabeta pruning. *Artificial Intelligence*, Vol. 6(4) pp. 293-326, 1975

[22] Y. Liang and J. Fugere. A neural network approach to mutual attribution between two opponents. *International Conference on Neural Networks and Brain* (ICNN&B '98), pp. 445-450, 1998

[23] G. Luger and W. Stubblefield. Artificial Intelligence: Structures and strategies for complex problem solving. Benjamin/Cummings, Redwood City, California, second edition, 1993

[24] J. Nash. Equilibrium points in n-person games. National Academy of Sciences. Vol. 36 pp. 48-49, 1950

[25] S. Parry. Evaluation of attrition methodologies for combat models, *Operations Research Department, Naval Postgraduate School Notes*, 1992

[26] W. Poundstone. The Prisoner's Dilemma. New York: Doubleday, 1993

[27] E. Rich and K. Knight. Artificial intelligence. McGraw-Hill, New York, second edition, 1991.

[28] S. Russell and P. Norvig. Artificial intelligence, Prenice Hall, Englewood Cliffs, New Jersey, 1997

[29] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, Vol. 3 pp. 210-229, 1959

[30] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, Vol. 53(2-3) pp. 273-290, 1992

[31] C. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, Vol. 41, pp. 256-275, 1950

[32] D. Slate and L. Atkin. Chess 4.5 – the Northwestern university chess program. *Chess Skill in Man and Machine*, Springer-Verlag, 1977

[33] A. Stentz. The focussed D* algorithm for real-time replanning. *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1652-1659, 1995

[34] C. Taylor. You ought to be in pixels. Time, April 12, 2004.

[35] J. Taylor. Lanchester models of warfare, *Operations Research Society*, Vols. 1 and 2, Arlington, Virginia, 1983

[36] A. Turing. Digital computers applied to games. In B. Bowden, editor, *Faster than Thought*, pp. 286-295. Pitman, 1953

[37] T. Turkes. Fitting Lanchester equations to the battle of Kursk data, Master Thesis, *Naval Postgraduate School*, Monterey, California, 2000

[38] S. Tzu. The Art of War, Oxford University Press, 1963

[39] S. Ulam and J. Metropolis. The Monte Carlo method. Journal of American Association, Vol. 44 pp. 335-345, 1949

[40] P. Winston. Artificial intelligence. Addison-Wesley, Reading, Massachusetts, third edition, 1992

[41] J. Yang and A. Gagarian. A fast approximation of homogeneous stochastic combat. *Naval Research Logistics*, Vol. 42 pp. 503-533, 1999